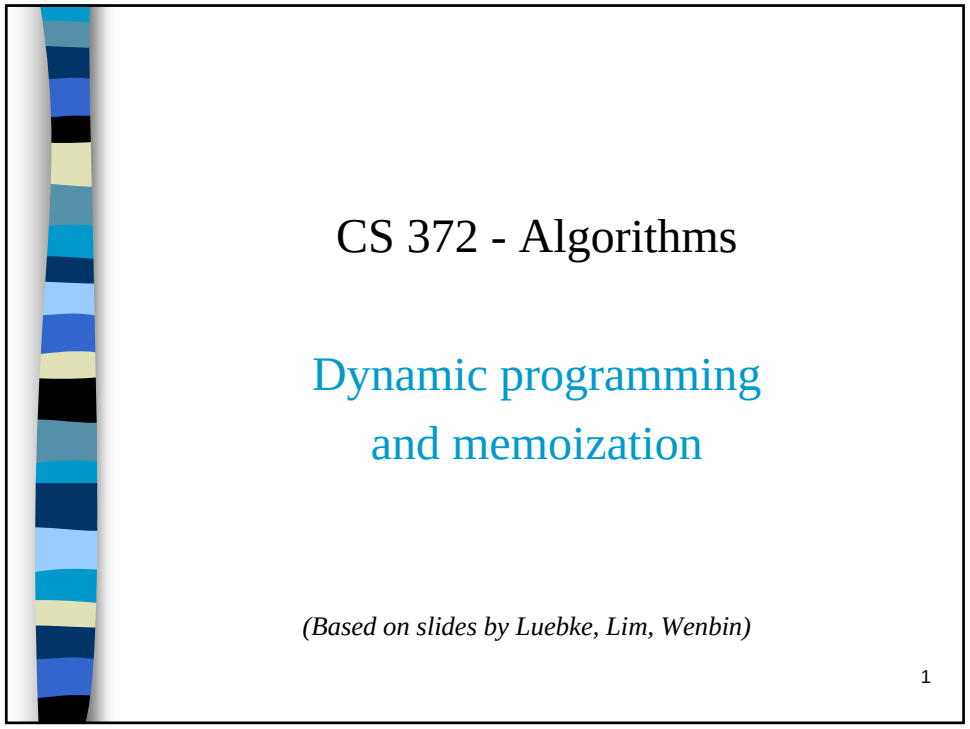
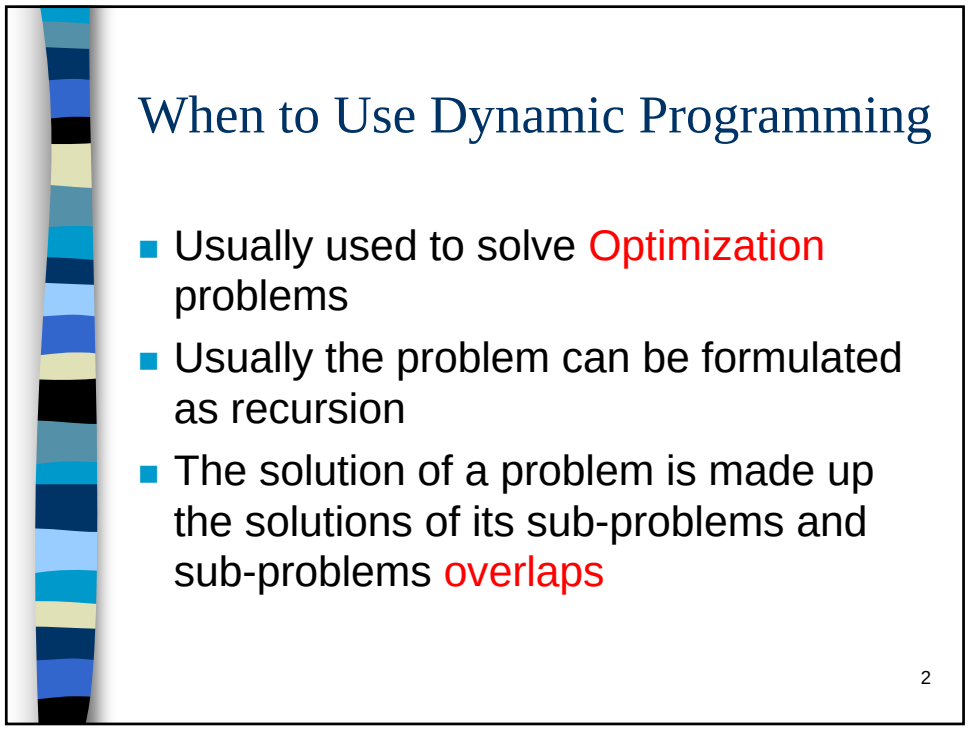


CS 372 - Algorithms

Dynamic programming and memoization

(Based on slides by Luebke, Lim, Wenbin)

1



When to Use Dynamic Programming

- Usually used to solve Optimization problems
- Usually the problem can be formulated as recursion
- The solution of a problem is made up the solutions of its sub-problems and sub-problems overlaps

2

E.g. Fibonacci Number

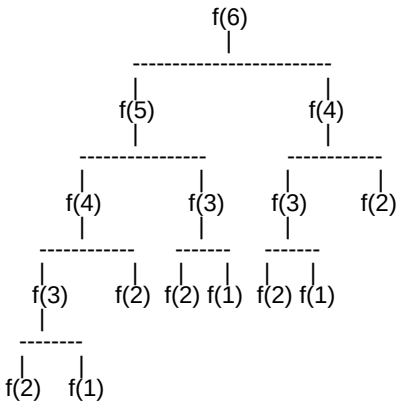
- The problem (Background)
  - Every mature pair of rabbits give birth to a pair of rabbits in every month
  - Newly born rabbits become mature after one month
  - There is one mature pair of rabbits at beginning of a year, how many pairs are there after one year?
- Formulation:
  - f(n) = f(n-1) + f(n-2);
  - f(1) = 1, f(2) = 2

3

FN: Naïve Solution

```
int f (int n) {
    int r;
    if (n<=2)
        then r = n;
        else r = f(n-1) + f(n-2);
    return r;
}
```

Tree of recursive calls for f(6):



4

Memoization

- Top-down approach
- Memorize whatever calculated, calculate only once

```
/* Memoization Method. Assume n is at most 100
 * long mem[101];      memset(mem, 0, sizeof(mem));
 */
int f (int n) {
    int r;
    if (mem[n] > 0) then return mem[n];
    if (n <= 2) then r = n;
                else r = f(n-1) + f(n-2);
    mem[n] = r;
    return r;
}
```

5

Turn Recursion into Memoization

Initialize memory in main function

```
int f () {
   if already calculated return the result
   calculate the result using recursion
   save the result in memory
   return the result
}
```

6

FN: Fill-in-the-table method

- Bottom-up approach
- Figure out the order in which the memory is filled in manually, fill in the table using loop

```
long mem[101];
int f (int n) {
    int i;
    mem[1] = 1; mem[2] = 2;
    for (i = 3; i <= n; i++)
        mem[i] = mem[i-1] + mem[i-2];
    return mem[n];
}
```

7

FN: Save Memory

- Only previous two results are needed in every step, so just keep these two result

```
int f (int n) {
    int pre_2, pre_1, cur, i;
    if (n <= 2) then return n;
    pre_2 = 1;  pre_1 = 2;
    for (i = 3; i <= n; i++) {
        cur = p_1 + p_2;    p_1 = cur;   p_2 = p_1;
    }
    return cur;
}
```

8

Dynamic programming

- It is used, when the solution can be recursively described in terms of solutions to subproblems (*optimal substructure*)
- Algorithm finds solutions to subproblems and stores them in memory for later use
- More efficient than "*brute-force methods*", which solve the same subproblems over and over again

9

Longest Common Subsequence (LCS)

Application: comparison of two DNA strings

Ex: X= {A B C B D A B }, Y= {B D C A B A}

Longest Common Subsequence:

X = A **B** **C** **B** D **A** B

Y = **B** D **C** A **B** **A**

Brute force algorithm would compare each subsequence of X with the symbols in Y

10

LCS Algorithm

- if |X| = m, |Y| = n, then there are 2^m subsequences of x; we must compare each with Y (n comparisons)
- So the running time of the brute-force algorithm is $O(n\,2^m)$
- Notice that the LCS problem has *optimal substructure*: solutions of subproblems are parts of the final solution.
- Subproblems: "find LCS of pairs of *prefixes* of X and Y"

11

LCS Algorithm

- First we'll find the length of LCS. Later we'll modify the algorithm to find LCS itself.
- Define X_i, Y_j to be the prefixes of X and Y of length i and j respectively
- Define $c[i,j]$ to be the length of LCS of X_i and Y_j
- Then the length of LCS of X and Y will be $c[m,n]$

$$c[i,j] = \begin{cases} c[i-1,j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i,j-1], c[i-1,j]) & \text{otherwise} \end{cases}$$

12

LCS recursive solution

$$c[i,j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- We start with $i = j = 0$ (empty substrings of x and y)
- Since X_0 and Y_0 are empty strings, their LCS is always empty (i.e. *c[0,0] = 0*)
- LCS of empty string and any other string is empty, so for every i and j: *c[0, j] = c[i,0] = 0*

13

LCS recursive solution

$$c[i,j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- When we calculate *c[i,j]*, we consider two cases:
- **First case:** *x[i]=y[j]*: one more symbol in strings X and Y matches, so the length of LCS X_i and Y_j equals to the length of LCS of smaller strings X_{i-1} and Y_{j-1} , plus 1

14

LCS recursive solution

$$c[i, j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- **Second case:** *x[i] != y[j]*
- As symbols don't match, our solution is not improved, and the length of LCS(X_i , Y_j) is maximum of LCS(X_i, Y_{j-1}) and LCS(X_{i-1},Y_j)

Why not just take the length of LCS(X_{i-1}, Y_{j-1}) ?

15

LCS: recursive solution

```
char a[1001], b[10001]; /* the two sequence */
int c (int i, int j) {
    int r;
    if (i == 0 || j == 0)  then r = 0;
        else if (a[i] == b[j])  then r = c (i-1, j-1) + 1;
                                else r = max(c(i, j-1), c(i-1, j));
    return r;
}
```

16

LCS: memoization

```
char a[1001], b[10001]; /* the two sequence */
int mem[1001][1001]; /* the memory */
memset(mem, -1, sizeof(mem)); /* do it in main method  */
int c (int i, int j) {
    int r;
    if (mem[i][j] >= 0) then return mem[i][j];
    if (i == 0 || j == 0) then r = 0;
        else if (a[i] == b[j]) then r = c (i-1, j-1) +1;
                                else r = max(c(i-1, j), c(i, j-1));
    mem[i][j] = r;
    return r;
}
```

17

LCS: Fill-in-the-table method

```
LCS-Length(X, Y)
1. m = length(X)  // get the # of symbols in X
2. n  = length(Y) // get the # of symbols in Y
3. for i = 1 to m    c[i,0] = 0     // special case: Y0
4. for j = 1 to n    c[0,j] = 0     // special case: X0
5. for i = 1 to m                   // for all Xi
6.      for j = 1 to n                        // for all Yj
7.          if ( Xi == Yj )
8.              then  c[i,j] = c[i-1,j-1] + 1
9.              else   c[i,j] = max( c[i-1,j], c[i,j-1] )
10. return c
```

18

LCS Example

We'll see how LCS algorithm works on the following example:

- X = ABCB
- Y = BDCAB

What is the Longest Common Subsequence of X and Y?

LCS(X, Y) = BCB
X = A **B** **C** **B**
Y = **B** D **C** A **B**

19

LCS Example (0)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	**B**	**D**	**C**	**A**	**B**
0	Xi						
1	**A**						
2	**B**						
3	**C**						
4	**B**						

X = ABCB; m = |X| = 4
Y = BDCAB; n = |Y| = 5
Allocate array c[5,4]

20

LCS Example (1)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**					
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

for i = 1 to m
for j = 1 to n

c[i,0] = 0
c[0,j] = 0

21

LCS Example (2)

ABCB
BDCAB

i		j	0	**1**	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**				
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

if (X_i == Y_j)
then c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

22

LCS Example (3)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**		
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

if (X_i == Y_j)
then c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

23

LCS Example (4)

ABCB
BDCAB

i		j	0	1	2	3	**4**	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	
2	**B**		**0**					
3	**C**		**0**					
4	**B**		**0**					

if (X_i == Y_j)
then c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

24

LCS Example (5)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1 →	1
2	B		0					
3	C		0					
4	B		0					

if (X_i == Y_j)
then c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

25

LCS Example (6)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1				
3	C		0					
4	B		0					

if (X_i == Y_j)
then c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

26

LCS Example (7)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	
3	C	0					
4	B	0					

if (X_i == Y_j)
then c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

27

LCS Example (8)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0					
4	B	0					

if (X_i == Y_j)
then c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

28

LCS Example (10)

ABCB
BDCAB

i		j	0	1	2	3	4	5
		Yj		**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**			
4	**B**		**0**					

if (X_i == Y_j)
then c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

29

LCS Example (11)

ABCB
BDCAB

i		j	0	1	2	3	4	5
		Yj		**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**		
4	**B**		**0**					

if (X_i == Y_j)
then c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

30

LCS Example (12)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0					

if (X_i == Y_j)
then c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

31

LCS Example (13)

ABCB
BDCAB

i		j	0	1	2	3	4	5
			Yj	B	D	C	A	B
0	Xi		0	0	0	0	0	0
1	A		0	0	0	0	1	1
2	B		0	1	1	1	1	2
3	C		0	1	1	2	2	2
4	B		0	1				

if (X_i == Y_j)
then c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

32

LCS Example (14)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	

if (X_i == Y_j)
then c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

33

LCS Example (15)

ABCB
BDCAB

i	j	0	1	2	3	4	5
		Yj	B	D	C	A	B
0	Xi	0	0	0	0	0	0
1	A	0	0	0	0	1	1
2	B	0	1	1	1	1	2
3	C	0	1	1	2	2	2
4	B	0	1	1	2	2	3

if (X_i == Y_j)
then c[i,j] = c[i-1,j-1] + 1
else c[i,j] = max(c[i-1,j], c[i,j-1])

34

LCS Algorithm Running Time

- LCS algorithm calculates the values of each entry of the array c[m,n]
- So what is the running time?

O(m*n)

since each c[i,j] is calculated in constant time, and there are m*n elements in the array

35

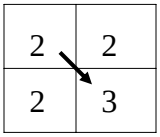
How to find actual LCS

- So far, we have just found the *length* of LCS, but not LCS itself.
- We want to modify this algorithm to make it output Longest Common Subsequence of X and Y

Each *c[i,j]* depends on *c[i-1,j]* and *c[i,j-1]*

or *c[i-1, j-1]*

For each c[i,j] we can say how it was acquired:



For example, here
c[i,j] = c[i-1,j-1] +1 = 2+1=3

36

How to find actual LCS - continued

- Remember that

$$c[i,j] = \begin{cases} c[i-1, j-1]+1 & \text{if } x[i] = y[j], \\ \max(c[i, j-1], c[i-1, j]) & \text{otherwise} \end{cases}$$

- So we can start from *c[m,n]* and go backwards
- Whenever *c[i,j] = c[i-1, j-1]+1*, remember *x[i]* (because *x[i]* is a part of LCS)
- When i=0 or j=0 (i.e. we reached the beginning), output remembered letters in reverse order

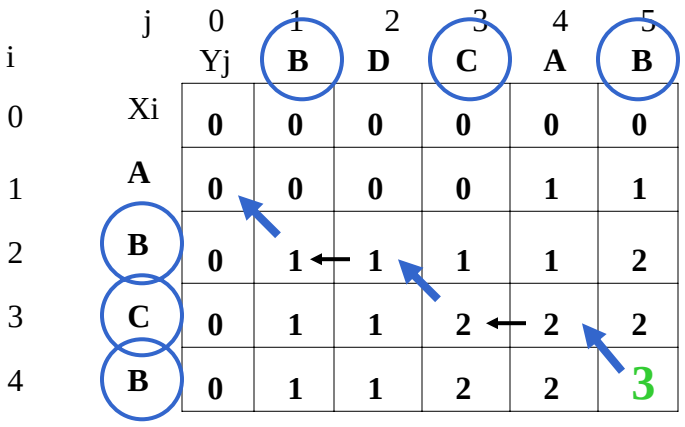
37

Finding LCS

i		j	0	1	2	3	4	5
			Yj	**B**	**D**	**C**	**A**	**B**
0	Xi		**0**	**0**	**0**	**0**	**0**	**0**
1	**A**		**0**	**0**	**0**	**0**	**1**	**1**
2	**B**		**0**	**1**	**1**	**1**	**1**	**2**
3	**C**		**0**	**1**	**1**	**2**	**2**	**2**
4	**B**		**0**	**1**	**1**	**2**	**2**	**3**

38

Finding LCS (2)



LCS (reversed order): **B C B**

LCS (straight order): **B C B**

(this string turned out to be a palindrome)

39

Exercises

Determine an LCS of

- science, student
- springtime, pioneer
- heroically, scholarly

40