## Dynamic Programming

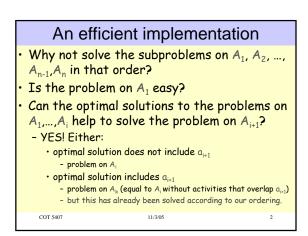
- Activity Problem Revisited: Given a set of n activities  $a_i = (s_i, f_i)$ , we want to schedule the maximum number of non-overlapping activities.
- New Approach:

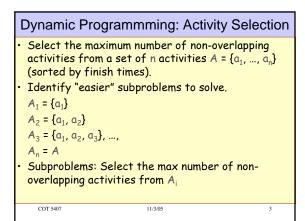
COT 5407

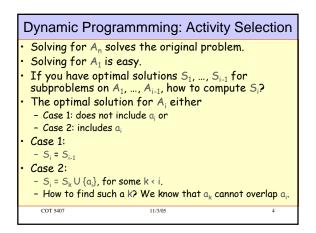
- Observation: To solve the problem on activities  $A = \{a_1, ..., a_n\}$ , we notice that either
  - optimal solution does not include an
  - then enough to solve subproblem on A<sub>n-1</sub>= {a<sub>1</sub>,...,a<sub>n-1</sub>}
     optimal solution includes a<sub>n</sub>

11/3/05

 Enough to solve subproblem on A<sub>k</sub> = {a<sub>1</sub>,...,a<sub>k</sub>}, the set A without activities that overlap a<sub>n</sub>.







Dynamic Programmming: A	Activity Selection
<ul> <li><u>DP-ACTIVITY-SELECTOR</u> (s, f 1. n = length[s] 2.N[1] = 1 // number of activiti 3.F[1] = 1 // last activity in S<sub>1</sub> 4. for i = 2 to n do 5. let k be the last activity finishe 6. if (N[i-1]&gt;N[k]) then // Case 7. N[i] = N[i-1] 8. F[i] = F[i-1]</li> </ul>	, es in S <sub>1</sub> ed before s <sub>i</sub>
9. else // Case 2 10. N[i] = N[k] + 1 11. F[i] = i	How to output S <sub>n</sub> ? Backtrack! Time Complexity? O(n lg n)
COT 5407 11/3/05	5

Dynamic Programming Features
<ul> <li>Identification of subproblems</li> </ul>
<ul> <li>Recurrence relation for solution of subproblems</li> </ul>
<ul> <li>Overlapping subproblems (sometimes)</li> </ul>
<ul> <li>Identification of a hierarchy/ordering of subproblems</li> </ul>
<ul> <li>Use of table to store solutions of subproblems (MEMOIZATION)</li> </ul>
<ul> <li>Optimal Substructure</li> </ul>

11/3/05

6

COT 5407

2

Longest Cor	mmon Subsequence
<b>S</b> <sub>1</sub> = CORIANDER	CORIANDER
S <sub>2</sub> = CREDITORS	CREDITORS
Longest Common Subseq	uence(S1[19], S2[19]) = <u>CRIR</u>
Subproblems:	
- LCS[S1[1i], S2[1j]	], for all i and j [BETTER]
<ul> <li>Recurrence Relatio</li> </ul>	n:
- LCS[i,j] = LCS[i-1, j	-1]+1, <u>if S<sub>1</sub>[i]=S<sub>2</sub>[j])</u>
LCS[i,j] = max { LCS	5[i-1, j], LCS[i, j-1] }, <u>otherwise</u>
• Table (m X n table)	
<ul> <li>Hierarchy of Solut</li> </ul>	ions?
,	
COT 5407	11/3/05 7

LCS Problem
LCS_Length (X, Y )
1. $m \leftarrow length[X]$
2. n ← Length[Y]
3. for i = 1 to m
4. do c[i, 0] ← 0
5. for j =1 to n
6. do c[0,j] ←0
7. for i = 1 to m
<ol> <li>do for j = 1 to n</li> </ol>
9. do if ( xi = yj )
10. then $c[i, j] \leftarrow c[i-1, j-1] + 1$
11. b[i, j] ← " )"
12. else if c[i-1, j] c[i, j-1]
13. then $c[i, j] \in c[i-1, j]$
14. b[i, j] ← "↑"
15. else
16. c[i, j] ← c[i, j-1]
17. b[i, j] ← "←"
18. return COT 5407 11/3/05 8
11/2/02 8

-	-	Н	A	В	I	Т	A	Т
	0	0	0	0	0	0	0	0
A	0	01	15	1	1	1	15	1←
L	0	01	11	11	11	11	11	11
P	0	01	11	11	1î	11	11	11
Н	0	15	11	11	1îî	11	11	11
A	0	11	25	2←	2←	24	25	2
В	0	11	21	35	3←	3	3←	3←
E	0	11	21	31	31	31	31	31
Т	0	11	21	31	31	45	4←	45



\_

\_

\_

## **Dynamic Programming vs. Divide-&-conquer**

- Divide-&-conquer works best when all subproblems are independent. So, pick partition that makes algorithm most efficient & simply combine solutions to solve entire problem. Dynamic programming is needed when subproblems are <u>dependent</u>; we don't know where to partition the problem. For example, let  $S_1$ = {ALPHABET}, and  $S_2$ = {HABITAT}. Consider the subproblem with  $S_1'$  = {ALPH},  $S_2'$  = {HABI}.
- Then, <u>LCS  $(S_1', S_2') + LCS (S_1-S_1', S_2-S_2') \neq LCS(S_1, S_2)</u>$ Divide-&-conquer is best suited for the case when no"overlapping subproblems" are encountered.</u>
- In dynamic programming algorithms, we typically solve each subproblem only once and store their solutions. But this is at the cost of space.

COT 5407 11/3/05 10

## Dynamic programming vs Greedy

 Dynamic Programming solves the sub-problems bottom up. The problem can't be solved until we find all solutions of sub-problems. The solution comes up when the whole problem appears.
 Greedy solves the sub-problems from top down. We first need to find the greedy choice for a problem, then reduce the problem to a smaller one. The solution is obtained when the whole problem disappears.
 Dynamic Programming has to try every possibility before solving the problem. It is much more expensive than greedy. However, there are some problems that greedy can not solve while dynamic programming can. Therefore, we first try greedy algorithm. If it fails then try dynamic programming.

COT 5407

11/3/05

11