COT 6405: Analysis of Algorithms

Giri NARASIMHAN

www.cs.fiu.edu/~giri/teach/6405F19.html

New Room Scheduling Problem

- Room Scheduling with Attendee Numbers: Given a set of requests to use a room (**with # of attendees**)
  - [1,4] (4), [3,5] (8), [0,6] (5), [5,7] (15), [3,8] (22), [5,9] (6), [6,10] (5), [8,11] (5), [8,12] (14), [2,13] (11), [12,14] (6)
- Schedule requests to **maximize the total # of attendees**
  - Greed is not good!

Dynamic Programming

- **Old 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.
- **General Approach**: Attempt a recursive solution

Recursive Solution

- **Observation**: To solve the problem on activities $A = \{a_1,\ldots,a_n\}$, we notice that either
  - optimal solution does not include a_n
    - then enough to solve subproblem on $A_{n-1} = \{a_1,\ldots,a_{n-1}\}$
  - optimal solution includes a_n
    - Enough to solve subproblem on $A_k = \{a_1,\ldots,a_k\}$, the set A without activities that overlap a_n.

Recursive Solution

int Rec-ROOM-SCHEDULING (s, f, t, n)

// Here n equals length[s];

// Input: first n requests with their s & f times & # attend

// It returns optimal number of requests scheduled

1. Let k be index of last request with finish time before s_n
2. Output larger of two values:
3. { Rec-ROOM-SCHEDULING (s, f, t, n-1),

   Rec-ROOM-SCHEDULING (s, f, t, k) + t[n] }

   // t[n] is number of attendees of n-th request

Dynamic Prog: Room Scheduling

- Let A be the set of n activities $A = \{a_1, \ldots, a_n\}$ (sorted by finish times).
- The inputs to the subproblems are:

  $A_1 = \{a_1\}$

  $A_2 = \{a_1, a_2\}$

  $A_3 = \{a_1, a_2, a_3\}, \ldots,$

  $A_n = A$
- i-th Subproblem: Select the max number of non-overlapping activities from A_i

An efficient implementation

- Why not solve the subproblems on A_1, A_2, …, A_{n-1}, A_n in that order?
- Is the problem on A_1 easy?
- Can the optimal solutions to the problems on $A_1,\ldots,A_i$ help to solve the problem on A_{i+1}?
  - YES! Either:
    - optimal solution does not include a_{i+1}
      - problem on A_i
    - optimal solution includes a_{i+1}
      - problem on A_k (equal to A_i without activities that overlap a_{i+1})
      - but this has already been solved according to our ordering.

Dynamic Prog: Room Scheduling

- Solving for A_n solves the original problem.
- Solving for A_1 is easy.
- If you have optimal solutions $S_1, \ldots, S_{i-1}$ for subproblems on $A_1, \ldots, A_{i-1}$, how to compute S_i?
- Recurrence Relation:
  - The optimal solution for A_i either
    - Case 1: does not include a_i or
    - Case 2: includes a_i
  - Case 1: $S_i = S_{i-1}$
  - Case 2: $S_i = S_k \cup \{a_i\}$, for some $k < i$.
    - How to find such a k? We know that a_k cannot overlap a_i.

DP: Room Scheduling w/ Attendees

- **DP-ROOM-SCHEDULING-w-ATTENDEES** (s, f, t)

```
n = length[s]
N[1] = t_1        // number of attendees in S_1
F[1] = 1          // last activity in S_1
for i = 2 to n do
    let k be the last activity finished before s_i
        if (N[i-1] > N[k] + t_i) then   // Case 1
            N[i] = N[i-1]
            F[i] = F[i-1]
        else    // Case 2
            N[i] = N[k] + t_i
            F[i] = i
Output N[n]
```

How to output S_n?
Backtrack!
Time Complexity?
O(n lg n)

Approach to DP Problems

- Write down a recursive solution
- Use recursive solution to identify list of **subproblems** to solve (there must be overlapping subproblems for effective DP)
- Decide a data structure to store solutions to subproblems (**MEMOIZATION**)
- Write down **Recurrence relation** for solutions of subproblems as suggested by the recursive sol
- Identify a **hierarchy/order** for subproblems
- Write down non-recursive solution/algorithm

Longest Common Subsequence

S_1 = CORIANDER   CORIANDER

S_2 = CREDITORS   CREDITORS

Longest Common Subsequence($S_1[1..9]$, $S_2[1..9]$)

= CRIR

Recursive Solution

LCS(S_1, S_2, m, n)

// m is length of S_1 and n is length of S_2

// Returns length of longest common subsequence

1. If (S_1[m] == S_2[n]), then
2. return 1 + LCS(S_1, S_2, m-1, n-1)
3. Else return larger of
4. LCS(S_1, S_2, m-1, n) and LCS(S_1, S_2, m, n-1)

Observation:

All the recursive calls correspond to subproblems to solve and they include LCS(S_1, S_2, i, j) for all i between 1 and m, and all j between 1 and n

Recurrence Relation & Memoization

- Recurrence Relation:
  - LCS[i,j] = LCS[i-1, j-1] + 1, <u>if S_1[i] = S_2[j])</u>

LCS[i,j] = max { LCS[i-1, j], LCS[i, j-1] }, <u>otherwise</u>

- Table (m X n table)
- Hierarchy of Solutions?
  - Solve in row major order

LCS Problem


```
LCS_Length (X, Y )
1. m ← 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
8.        do for j = 1 to n
9.            do if ( xi = yj )
10.                 then c[i, j] ← 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] ← c[i-1, j]
14.                        b[i, j] ← "↑"
15.                     else
16.                        c[i, j] ← c[i, j-1]
17.                        b[i, j] ← "←"
18. return  c[m,n]
```

LCS Example

		H	A	B	I	T	A	T
	0	0	0	0	0	0	0	0
A	0	0↑	1↖	1←	1←	1←	1↖	1←
L	0	0↑	1↑	1↑	1↑	1↑	1↑	1↑
P	0	0↑	1↑	1↑	1↑	1↑	1↑	1↑
H	0	1↖	1↑	1↑	1↑	1↑	1↑	1↑
A	0	1↑	2↖	2←	2←	2←	2↖	2←
B	0	1↑	2↑	3↖	3←	3←	3←	3←
E	0	1↑	2↑	3↑	3↑	3↑	3↑	3↑
T	0	1↑	2↑	3↑	3↑	4↖	4←	4↖

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 **dependent**; 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, $LCS(S_1', S_2') + LCS(S_1-S_1', S_2-S_2') \neq LCS(S_1, S_2)$
- Divide-&-conquer is best suited for the case when no "overlapping subproblems" are encountered.
- In dynamic programming algorithms, we typically solve each subproblem only once and store their solutions. But this is at the cost of space.

Dynamic programming vs Greedy

1. 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.

2. 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.

Fractional Knapsack Problem

- Burglar's choices:

  Items: $x_1, x_2, \ldots, x_n$

  Value: $v_1, v_2, \ldots, v_n$

  Max Quantity: $q_1, q_2, \ldots, q_n$

  Weight per unit quantity: $w_1, w_2, \ldots, w_n$

  Getaway Truck has a weight limit of B.

  Burglar can take "fractional" amount of any item.

  How can burglar maximize value of the loot?
- Greedy Algorithm works!

  Pick the maximum possible quantity of highest value per weight item. Continue until weight limit of truck is reached.

0-1 Knapsack Problem

- Burglar's choices:
  Items: $x_1, x_2, \ldots, x_n$
  Value: $v_1, v_2, \ldots, v_n$
  Weight: $w_1, w_2, \ldots, w_n$
  Getaway Truck has a weight limit of B.
  Burglar cannot take "fractional" amount of item.
  How can burglar maximize value of the loot?
- Greedy Algorithm does not work! Why?
- Need dynamic programming!

0-1 Knapsack Problem

- Subproblems?
  - V[j, L] = Optimal solution for knapsack problem assuming a truck of weight limit L and choice of items from set {1,2,…, j}.
  - V[n, B] = Optimal solution for original problem
  - V[1, L] = easy to compute for all values of L.
- Table of solutions?
  - V[1..n, 1..B]
- Ordering of subproblems?
  - Row-wise
- Recurrence Relation? [Either x_j included or not]
  - $V[j, L] = \max \{ V[j-1, L], \quad v_j + V[j-1, L-w_j] \}$

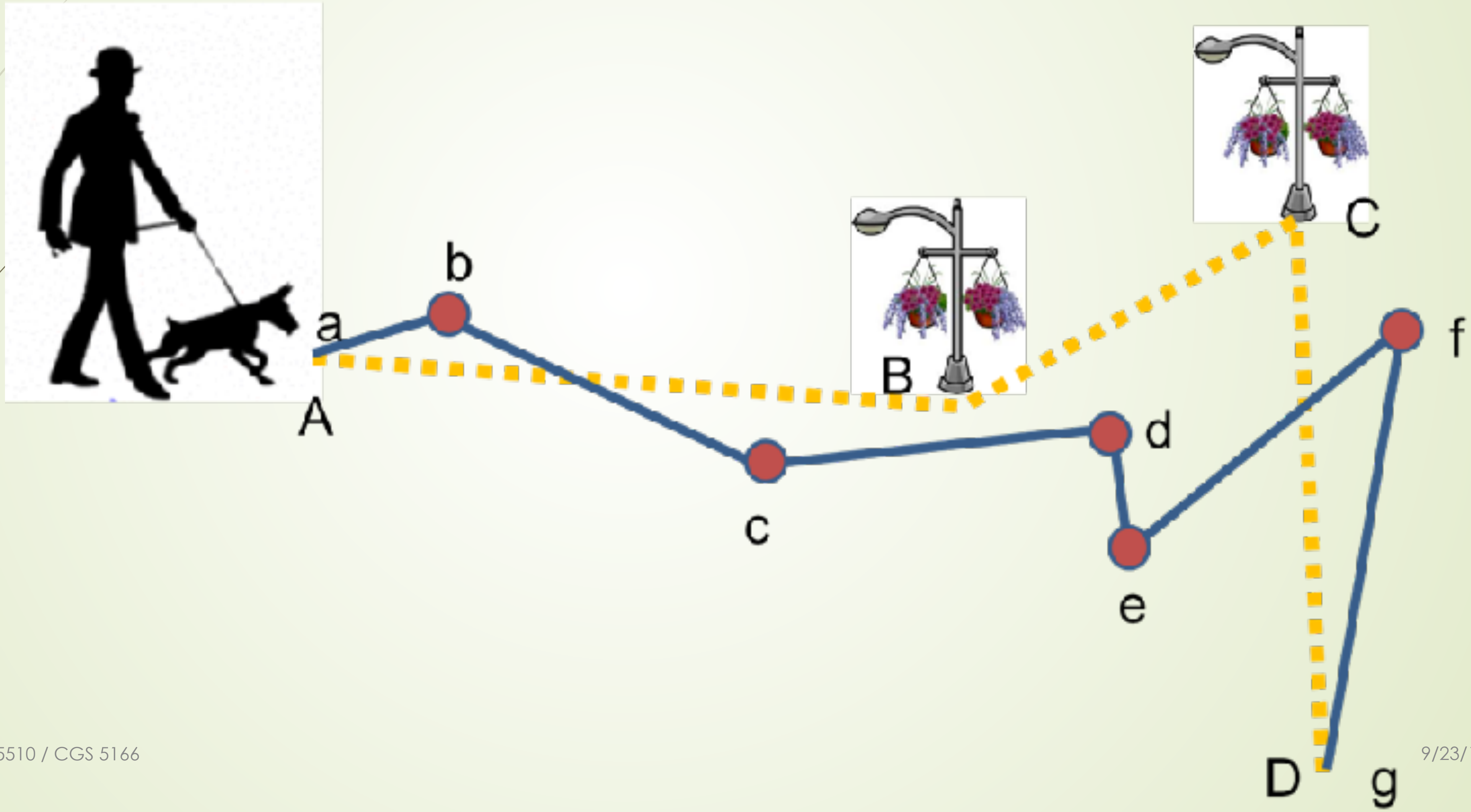
1-d, 2-d, 3-d Dynamic Programming

- Classification based on the dimension of the table used to store solutions to subproblems.
- 1-dimensional DP
  - Activity Problem
- 2-dimensional DP
  - LCS Problem
  - 0-1 Knapsack Problem
  - Matrix-chain multiplication
- 3-dimensional DP
  - All-pairs shortest paths problem

Matrix Chain Product

- MCP[1,n] = Min
  - MCP[1,k] + MCP[k+1,n] + cost(1,k,n)
  - Since we don’t know the value of k
    - We try every possible value of k

Shortest Leash Problem



Shortest Leash Problem … 1

- L[k,j] = shortest leash for a walk from start to k-th stop for dog walker and j-th stop for dog
- L[k,j] = Min of 2 possibilities
  - Max{ L[k-1, j-1], ssd[k-1, j-1]}
  - Max{L[k-1, j], spd[k-1, j]}
  - Max{L[k, j-1], psd[k, j-1]}