COT 5407: Introduction to Algorithms

Giri Narasimhan
ECS 254A; Phone: x3748
giri@cis.fiu.edu

http://www.cis.fiu.edu/~giri/teach/5407S17.html

https://moodle.cis.fiu.edu/v3.1/course/view.php?id=1494

Room Scheduling Problem

- **Room Scheduling**: Given a set of requests to use a room
  - [1,4], [3,5], [0,6], [5,7], [3,8], [5,9], [6,10], [8,11], [8,12], [2,13], [12,14]
  - Schedule largest number of above requests in the room
- **Greedy Algorithm worked!**
  - Sort by finish time and pick in "greedy" fashion
- Now let's modify the 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
  - Greedy Solution will be [1,4], [5,7], [8,11], [12,14]
  - And will satisfy 4 + 15 + 5 + 6 = 30 attendees
  - Greed is not good!

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.
- General Approach: Attempt a recursive solution

Recursive Solution

- Observation: To solve the problem on activities $A = \{a_1,\dots,a_n\}$, we notice that either
  - optimal solution does not include a_n
    - then enough to solve subproblem on $A_{n-1} = \{a_1,\dots,a_{n-1}\}$
  - optimal solution includes a_n
    - Enough to solve subproblem on $A_k = \{a_1,\dots,a_k\}$, 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, n-1),

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

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

Observations

- If we look at all subproblems generated by the recursive solution, and ignore repeated calls, then we see the following calls:
  - **Rec-ROOM-SCHEDULING (s, f, n-1)**
    - **Rec-ROOM-SCHEDULING (s, f, n-2)**
      - …
    - **Rec-ROOM-SCHEDULING (s, f, n')**
      - …
  - **Rec-ROOM-SCHEDULING (s, f, k)**
    - **Rec-ROOM-SCHEDULING (s, f, k-1)**
      - …
    - **Rec-ROOM-SCHEDULING (s, f, k')**
      - …
- Above list includes all subproblems **Rec-ROOM-SCHEDULING (s, f, i)** for all values of i between 1 and n

Dynamic Programming: 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,…,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 Programming: 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)

1. n = length[s]
2. N[1] = t_1 // number of attendees in S_1
3. F[1] = 1 // last activity in S_1
4. **for** i = 2 **to** n **do**
5. let k be the last activity finished before s_i
6. **if** (N[i-1] > N[k] + t_i) **then** // Case 1
7. N[i] = N[i-1]
8. F[i] = F[i-1]
9. **else** // Case 2
10. N[i] = N[k] + t_i
11. F[i] = I
12. 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
- 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:
  - $LCS[i,j] = LCS[i-1, j-1] + 1$, if $S_1[i] = S_2[j]$)

    $LCS[i,j] = \max \{ LCS[i-1, j], LCS[i, j-1] \}$, otherwise
- Table (m X n table)
- Hierarchy of Solutions?
  - Solve in row major order

LCS Problem

```
LCS_Length (X, Y )
m ← length[X]
n ← Length[Y]
for i = 1 to m
do c[i, 0] ← 0
for j =1 to n
do c[0,j] ←0
for i = 1 to m
        do for j = 1 to n
            do if ( xi = yj )
                then c[i, j] ← c[i-1, j-1] + 1
                    b[i, j] ← " ↖"
                else if c[i-1, j] c[i, j-1]
                        then c[i, j] ← c[i-1, j]
                        b[i, j] ← "↑"
                    else
                        c[i, j] ← c[i, j-1]
                        b[i, j] ← "←"
return  c[m,n]
```