COT 6405: Analysis of Algorithms

Giri NARASIMHAN

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

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

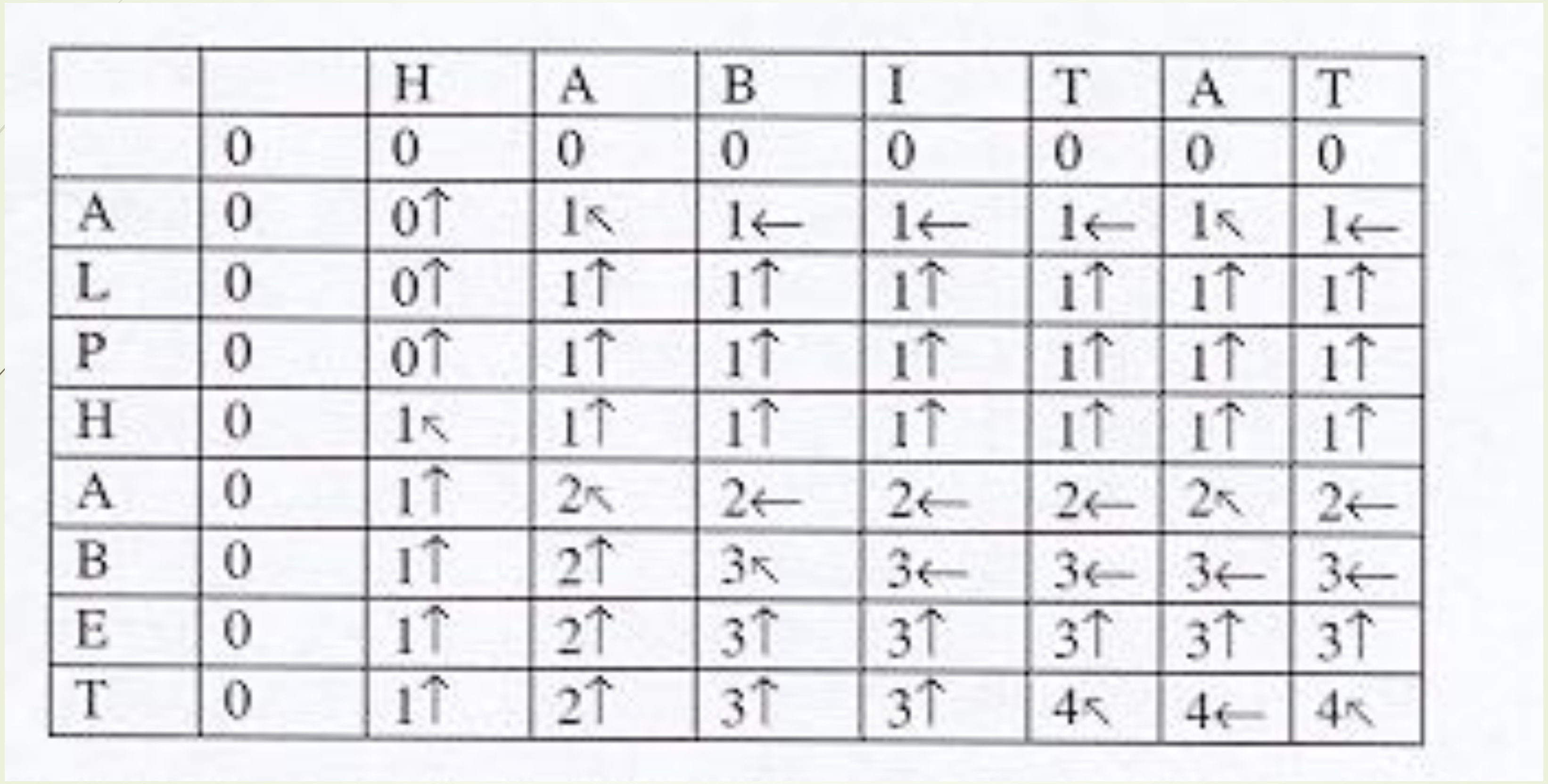
- Recurrence Relation:
  - LCS[i,j] = LCS[i-1, j-1] + 1, <u>if S_1[i] = S_2[j])</u>

  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 = 0 to m
4. do c[i, 0] ← 0
5. for j =0 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, $\underline{LCS(S_1', S_2') + LCS(S_1 - S_1', S_2 - S_2')} \neq \underline{LCS(S_1, S_2)}$
- 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: $x_1, x_2, \dots, x_n$

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

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

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

  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

Amortized Analysis

Amortized Analysis

- Consider (worst-case) time complexity of sequence of n operations, not cost of a single operation.
- *Traditional Analysis:* Cost of sequence of n operations = n S(n), where S(n) = worst case cost of each of the n operations
- *Amortized Cost* = T(n)/n, where T(n) = worst case total cost of n operations in the sequence.
- Amortized cost can be small even with some expensive operations. Worst case may not occur in every operation, even in worst case. Cost of operations often correlated.

Problem 1: Binary Counter

- Data Structure: binary counter b.
- Operations: Inc(b).
  - Cost of Inc(b) = number of bits flipped in the operation.
- What's the total cost of N operations when this counter counts up to integer N?
- *Approach 1: simple analysis*
  - Size of counter is log(N). Worst case when every bit flipped. For N operations, total worst-case cost = O(Nlog(N))

Approach 2: Binary Counter

- Intuition: **Worst case cannot happen all the time!**

000000

00000**1**

0000**10**

00001**1**

000**100**

00010**1**

0001**10**

00011**1**

Bit 0 flips every time;

Bit 1 flips every other time;

Bit 2 flips every fourth time, etc…

Bit k flips every 2^k time.

So the total bits flipped in N operations, when the counter counts from 1 to N, will be = ?

$$T(N) = \sum_{k=0}^{\log N} \frac{N}{2^k} < N \sum_{k=0}^{\infty} \frac{1}{2^k} = 2N$$

So the amortized cost will be T(N)/N = 2.

Approach 3: Binary Counter

- For k bit counters, the total cost is
  t(k) = 2 x t(k-1) + 1
- So for N operations, T(N) = t(log(N)).
- t(k) = ?
- T(N) can be proved to be bounded by 2N.

Amortized Analysis: Potential Method

- For n operations, the data structure goes through states: D_0, D_1, D_2, …, D_n with costs c_1, c_2, …, c_n
- Define potential function $\Phi(D_i)$: represents the <u>potential energy</u> of data structure after i_{th} operation.
- The amortized cost of the i_{th} operation is defined by:

$$\boxed{\hat{c}_i = c_i + \Phi(D_i) - \Phi(D_{i-1})}$$

- The total amortized cost is

$$\sum_{i=1}^{n} \hat{c}_i = \sum_{i=1}^{N} \left(c_i + \Phi(D_i) - \Phi(D_{i-1})\right) = \Phi(D_n) - \Phi(D_0) + \sum_{i=1}^{n} c_i$$

$$\sum_{i=1}^{n} c_i = -\left(\Phi(D_n) - \Phi(D_0)\right) + \sum_{i=1}^{n} \hat{c}_i$$

Potential Method for Binary Counter

- Potential function = ??
- $\Phi(D)$ = # of 1's in counter
- Assume that in i-th iteration Inc(b) changes
  - $1 \rightarrow 0$ (j bits)
  - $0 \rightarrow 1$ (1 bit)
  - $\Phi(D_{i-1}) = k$; $\Phi(D_i) = k - j + 1$
  - Change in potential = $(k - j + 1) - k = 1-j$
  - Real cost = $j + 1$
  - Amortized cost = Real cost + change in potential
  - Amortized cost = $j + 1 - j + 1 = 2$

Problem 2: Stack Operations

- Data Structure: Stack
- Operations:
  - *Push(s,x)* : Push object *x* into stack *s*.
    - Cost: T(push) = O(1).
  - *Pop(s) :* Pop the top object in stack *s*.
    - Cost: T(pop) = O(1).
  - *MultiPop(s,k) ;* Pop the top *k* objects in stack *s*.
    - Cost: T(mp) = O(size(s)) worst case
- *Assumption:* Start with an empty stack
- *Simple analysis:* For N operations, maximum stack size = N. Worst-case cost of *MultiPop* = O(N). Total worst-case cost of N operations is at most N x T(mp) = O(N^2).

Amortized analysis: Stack Operations

- Intuition: Worst case cannot happen all the time!
- Idea: pay a dollar for every operation, then count carefully.
- Pay \$2 for each *Push* operation, one to pay for operation, another for "future use" (pin it to object on stack).
- For *Pop* or *MultiPop*, instead of paying from pocket, pay for operations with extra dollar pinned to popped objects.
- Total cost of N operations must be less than 2 x N
- Amortized cost = T(N)/N = 2.

Potential Method for Stack Problem

- Potential function $\Phi(D)$ = # of items in stack
- Push
  - Change in potential = 1; Real cost = 1
  - Amortized Cost = 2
- MultiPop [Assume j items popped in ith iter]
  - $\Phi(D_{i-1}) = k$; $\Phi(D_i) = k - j$
  - Real cost = j
  - Change in potential = -j
  - Amortized cost = Real cost + change in potential
  - Amortized cost = j - j = 0

Pop: j = 1