Dynamic Programming Features

- Identification of subproblems
- Recurrence relation for solution of subproblems
- Overlapping subproblems (sometimes)
- Identification of a hierarchy/ordering of subproblems
- Use of table to store solutions of subproblems (MEMOIZATION)
- Optimal Substructure

Longest Common Subsequence

S_1 = CORIANDER   CORIANDER

S_2 = CREDITORS   CREDITORS

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

Subproblems:

- LCS[$S_1[a..b]$, $S_2[c..d]$], for all a, b, c, and d
- LCS[$S_1[1..i]$, $S_2[1..j]$], for all i and j [BETTER]

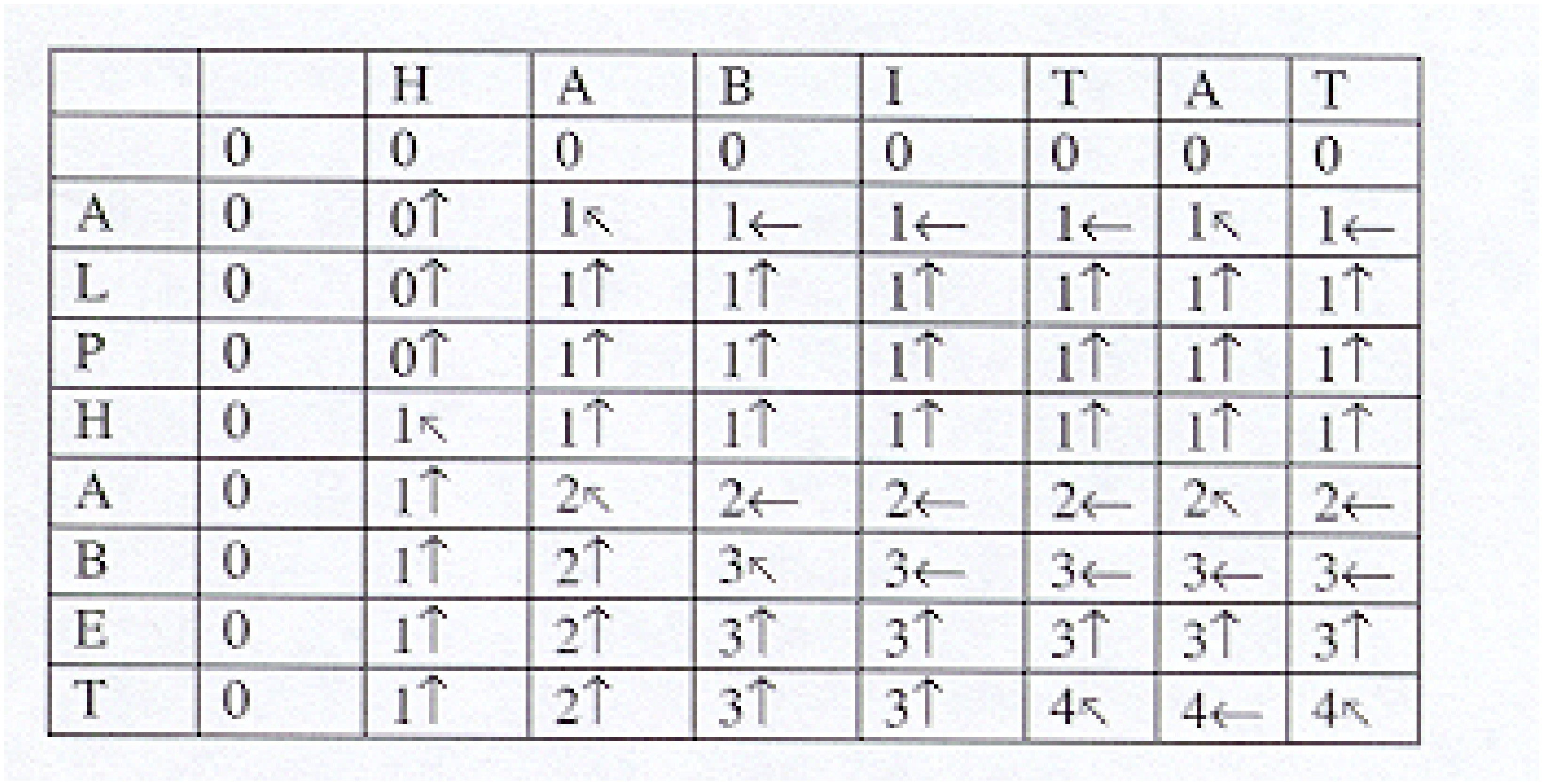
- Recurrence Relation:
  - LCS[i,j] = LCS[i-1, j-1] + 1, if $S_1[i] = S_2[j]$)

    LCS[i,j] = max { LCS[i-1, j], LCS[i, j-1] }, otherwise
- Table (m X n table)
- Hierarchy of Solutions?

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
```

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 the partition that makes the algorithm most efficient. Then simply combine their solutions to solve the entire problem.
- Dynamic programming is needed when subproblems are dependent and 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, $\underline{\text{LCS}(S_1', S_2') + \text{LCS}(S_1-S_1', S_2-S_2') \neq \text{LCS}(S_1, S_2)}$
- 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 algorithms are less efficient than greedy algorithms because they typically try every possible way of partitioning the problem. However, there are many problems that greedy methods cannot solve while dynamic programming can. Therefore, we first try greedy algorithm. If it fails then try dynamic programming.
2. Dynamic Programming solutions using memoization are solved bottom-up. Dynamic Programming solutions using recursion are solved top-down (less efficient). Solution of the problem requires solutions of all subproblems.
3. Greedy algorithms are neither top-down nor bottom-up. They are "incremental" solutions and rely on the assumption that the greedy choice is a good choice. Making the greedy choice reduces the problem to a smaller subproblem.

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]
  - $V[j, L] = \max \{ V[j-1, L],$
    $v_j + V[j-1, L-w_j] \}$