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

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

Graphs

- Graph $G(V,E)$
- V Vertices or Nodes
- E Edges or Links: pairs of vertices
- D Directed vs. Undirected edges
- Weighted vs Unweighted
- Graphs can be augmented to store extra info (e.g., city population, oil flow capacity, etc.)
- Paths and Cycles
- Subgraphs $G'(V',E')$, where V' is a subset of V and E' is a subset of E
- Trees and Spanning trees

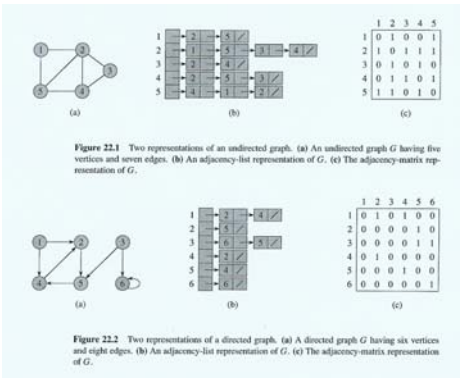

Figure 22.1 Two representations of an undirected graph. **(a)** An undirected graph G having five vertices and seven edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.

Figure 22.2 Two representations of a directed graph. **(a)** A directed graph G having six vertices and eight edges. **(b)** An adjacency-list representation of G. **(c)** The adjacency-matrix representation of G.