BITS, PILANI – K. K. BIRLA GOA CAMPUS

Design & Analysis of Algorithms

(CS F364)

Lecture No. 4

Re Cap - Dynamic Programming

- **Main idea:**
  - set up a ***recurrence*** relating a solution to a larger instance to solutions of some smaller instances
  - solve smaller instances once
  - record solutions in a table
  - extract solution to the initial instance from that table

Examples : Fibonacci Sequence & Binomial Coefficient

Coin Change Problem

Coin Change Problem

Given a value N, if we want to make change for N paisa, and we have infinite supply of each of C = { 1, 5, 10, 25} valued coins, what is the minimum number of coins to make the change?

Solution – Easy (We all know this)

Greedy Solution

Coin Change Problem

Suppose we want to compute the minimum number of coins with values

d[1], d[2], …,d[n] where each d[i]>0

& where coin of denomination i has value d[i]

Let c[i][j] be minimum number of coins required to pay an amount of j units 0<=j<=N using only coins of denomination s 1 to i, 1<=i<=n

C[n][N] is the solution to the problem

Coin Change Problem

In calculating c[i][j], notice that:

- Suppose **we do not use** the coin with value d[i] in the solution of the (i,j)-problem,

  then **c[i][j] = c[i-1][j]**
- Suppose **we use** the coin with value d[i] in the solution of the (i,j)-problem,

  then **c[i][j] = 1 + c[i][j-d[i]]**

Since we want to minimize the number of coins, we choose whichever is the better alternative

Coin Change Problem – Recurrence

Therefore

c[i][j] = min{c[i-1][j], 1 + c[i][j-d[i]]}

&

c[i][0] = 0 for every i

Alternative 1

Recursive algorithm

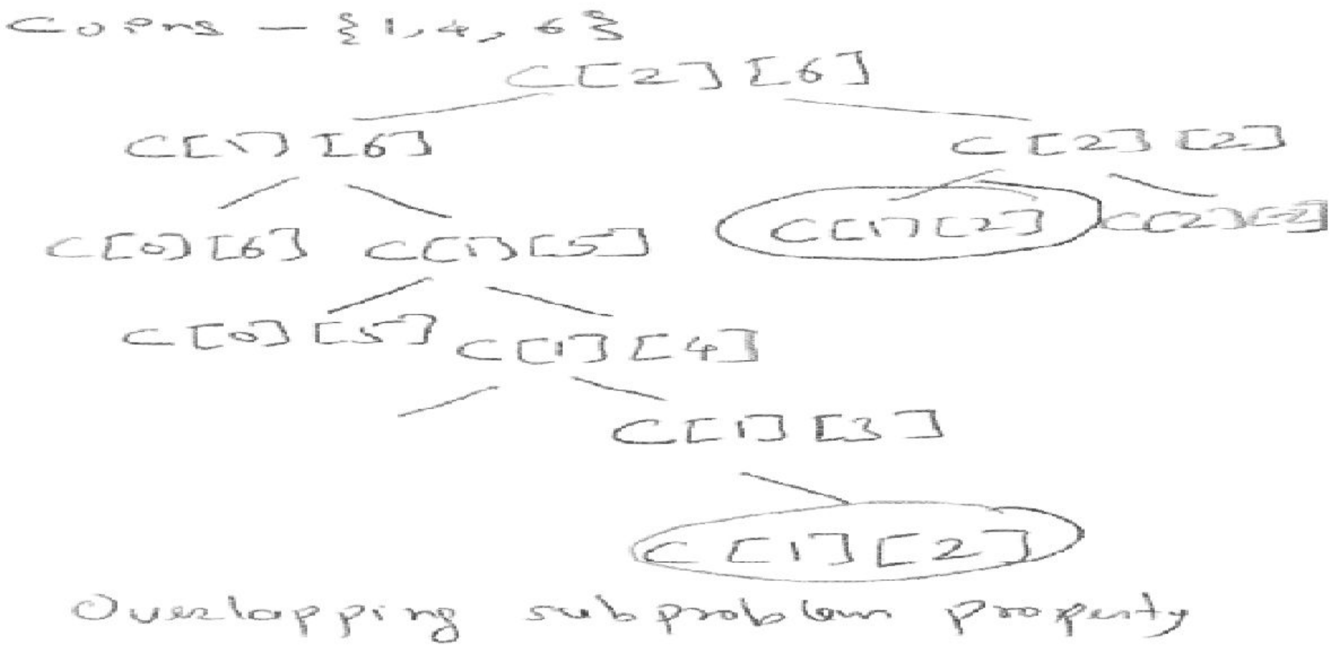
When a recursive algorithm revisits the same problem over and over again, we say that the optimization problem has **overlapping subproblems.**

How to observe/prove that problem has overlapping subproblems.

Answer – **Draw Computation tree and observe**

Overlapping Subproblems

Computation Tree



Dynamic-programming **algorithms typically take advantage of** **overlapping subproblems** **by solving each subproblem once and then storing the solution in a table where it can be looked up when needed, using constant time per lookup.**

Coin Change Problem

- **Example**

We have to pay 8 units with coins worth 1,4 & 6 units

For example c[2][6] is obtained in this case as the smaller of c[1][6] and 1+ c[2][6-d[2]] = 1+c[2][2]

The other entries of the table are obtained similarly

	0	1	2	3	4	5	6	7	8
d[1]=1	0	1	2	3	4	5	6	7	8
d[2]=4	0	1	2	3	1	2	3	4	2
d[3]=6	0	1	2	3	1	2	1	2	2

The table gives the solution to our problem for all the instances involving a payment of 8 units or less

Analysis

Time Complexity

We have to compute **n(N+1)** entries

Each entry takes **constant time** to compute

Running time – **O(nN)**

Question

- How can you modify the algorithm to actually compute the change (i.e., the multiplicities of the coins)?
- Modify the algorithm to handle exceptional cases.

Optimal Substructure Property

- **Why does the solution work?**

Optimal Substructure Property/ Principle of Optimality

- ***The optimal solution to the original problem incorporates optimal solutions to the subproblems.***
- ***In an optimal sequence of decisions or choices, each subsequence must also be optimal***

This is a hallmark of problems amenable to dynamic programming.

- Not all problems have this property.

Optimal Substructure Property

- In our example though we are interested only in c[n][N], we took it granted that all the other entries in the table must also represent optimal choices.
- If c[i][j] is the optimal way of making change for j units using coins of denominations 1 to I, then c[i-1][j] & c[i][j-d[i]] must also give the optimal solutions to the instances they represent

Optimal Substructure Property

How to prove **Optimal Substructure Property?**

Generally by Cut-Paste Argument or By Contradiction

Note

Optimal Substructure Property looks obvious

But it does not apply to every problem.

Exercise:

Give an problem which does not exhibit Optimal Substructure Property.

Dynamic Programming Algorithm

The **dynamic-programming** algorithm can be broken into a sequence of **four steps.**

1. Characterize the structure of an optimal solution.
   Optimal Substructure Property
2. **Recursively define** the value of an optimal solution.
3. Compute the value of an optimal solution in a bottom-up fashion.
   Overlapping subproblems
4. Construct an optimal solution from computed information.
   (not always necessary)