Dynamic Programming Algorithms
Greedy Algorithms
Lecture 27

Return to Divide-and-Conquer
• Divide-and-Conquer
  – Divide big problem into smaller subproblems
  – Conquer each subproblem separately
  – Merge the solutions of the subproblems into the solution of the big problem

  Top-down approach

  Example:

  Fibonacci(n)

  if (n ≤ 1) then return n
  else return Fibonacci(n-1) + Fibonacci(n-2)

  Very slow algorithm because we recompute Fibonacci(i) many many times...

Dynamic programming

• Solve each small problem once, saving their solution
• Use the solutions of small problems to obtain solutions to larger problems

FibonacciDynProg(n)

int F[0...n];
F[0] = 0 ;
F[1] = 1;
for i = 2 to n do  // compute min{ Opt( i - C[j] ) | j = 1...k, C[j] ≤ n }
m = +∞
for j = 0 to k-1 do
  if (C[j] ≤ i) then mini = min(mini, Opt[ i-C[j] ] )
Opt[i] = 1 + mini
return Opt[n]

Bottom-up approach

The change making problem

• A country has coins worth 1, 3, 5, and 8 cents
• What is the smallest number of contains needed to make
  – 25 cents?
  – 15 cents?
• In general, with coins denominations C_1, C_2, ..., C_k, how to find the smallest number of coins needed to make a total of n cents?

Dyn. Prog. Algo. for making change

• Let Opt(n) be the optimal number of coins needed to make n cents
• We write a recursive formula for Opt(n):
  Opt(0) = 0
  Opt(n) = 1 + min{ Opt( n - C_j ) | j = 1...k, C_j ≤ n }

Example: with coins 1, 3, 5, 8
= 1 + min{ 3, 3, 2, 3 }
= 1 + 2 = 3

Algorithm makeChange(C[0..k-1], n)
Input: an array C containing the values of the coins
an integer n
Output: The minimal number of coins needed to make a total of n

int Opt[] = new int[n+1];  // Opt[0...n]
Opt[0] = 0
for i =1 to n do  // compute min{ Opt( i - C[j] ) | j = 1...k, C[j] ≤ n }
m = +∞
for j = 0 to k-1 do
  if (C[j] ≤ i) then mini = min(mini, Opt[ i-C[j] ] )
Opt[i] = 1 + mini
return Opt[n]
Making change - Greedy algorithm

• You need to give x¢ in change, using coins of 1, 5, 10, and 25 cents. What is the smallest number of coins needed?

• Greedy approach:
  – Take as many 25¢ as possible, then
  – take as many 10¢ as possible, then
  – take as many 5¢ as possible, then
  – take as many 1¢ as needed to complete

• Example: 99¢ = 3*25¢ + 2*10¢ + 1*5¢ + 4*1¢

• Is this optimal?

Greedy-choice property

• A problem has the greedy choice property if:
  – An optimal solution can be reached by a series of locally optimal choices

• Change making: 1, 5, 10, 25¢: greedy is optimal
  1, 6, 10¢: greedy is not optimal

• Most optimization problems, greedy algorithm are not optimal. However, when they are, they are usually the fastest available.

Longest Increasing Subsequence

Problem: Given an array A[0..n] of integers, find the longest increasing subsequence in A.

Example: A = 5 1 4 2 8 4 9 1 8 9 2

Solution:

Slow algorithm: Try all possible subsequences…

for each possible subsequences s of A do
  if (s is in increasing order) then
    if (s is best seen so far) then save s

return best seen so far

Dynamic Programming Solution

Let LIS[i] = length of the longest increasing subsequence ending at position i and containing A[i].

A = 5 1 4 2 8 4 9 1 8 9 2

LIS =

LIS[0] = 1

Dynamic Programming Solution

Algorithm LongestIncreasingSubsequence(A, n)
Input: an array A[0...n] of numbers
Output: the length of the longest increasing subsequence of A

LIS[0] = 1
for i = 1 to n do
  LIS[i] = -1
  for j = 0 to i-1 do
  return max(LIS)

Dynamic Programming Framework

• Dynamic Programming Algorithms are mostly used for optimization problems

• To be able to use Dyn. Prog. Algo., the problem must have certain properties:
  – Simple subproblems: There must be a way to break the big problem into smaller subproblems. Subproblems must be identified with just a few indices.
  – Subproblem optimization: An optimal solution to the big problem must always be a combination of optimal solutions to the subproblems.
  – Subproblem overlap: Optimal solutions to unrelated problems can contain subproblems in common.
Greedy-choice property

- A problem has the greedy choice property if:
  - An optimal solution can be reached by a series of locally optimal choices
- Change making: 1, 5, 10, 25 ¢: greedy is optimal
  1, 6, 10 ¢: greedy is not optimal

- Most optimization problems, greedy algorithm are not optimal. However, when they are, they are usually the fastest available.