CS483 Analysis of Algorithms
Lecture 07 – Dynamic Programming 01 *

Jyh-Ming Lien

March 26, 2009

*this lecture note is based on *Algorithms* by S. Dasgupta, C.H. Papadimitriou, and U.V. Vazirani and *Introduction to the Design and Analysis of Algorithms* by Anany Levitin.

Dynamic Programming

- A term coined by Richard Bellman in the 1940s

(Image from ieee.org. Richard Bellman, 1920 - 1984)

- Some problems solved by dynamic programming
  - Longest increasing subsequences
  - Fibonacci number
  - Knapsack problem
  - All-pairs shortest path problem (Floyd's algorithm)
  - Optimal binary search tree problem
  - Multiplying a sequence of matrices
  - String matching (or DNA sequence matching), where we search for the string closest to the pattern
  - Convex decomposition of polygons
  - ...

A Toy Example: Fibonacci number (again)

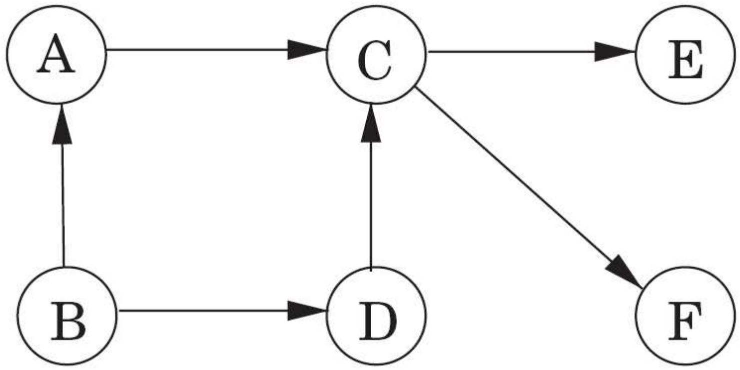
- $f(n) = f(n-1) + f(n-2), f(0) = 1, f(1) = 1$
- Recursive brute force approach:

- DP approach:

- What's the difference?
- What's the difference between divide-n-conquer and dynamic programming?

Shortest path in DAGs

- Example:



- Algorithm

Algorithm 0.1: DAG-SHORTEST-PATH(G, s)

Longest increasing subsequences

- Given a sequence of integers, find the longest *increasing* sequence.
- Example 1: 5, 2, 8, 6, 3, 6, 9, 7 (the longest increasing subsequences is: 2, 3, 6, 9)
- How do we solve this problem using dynamic programming?
- Key observation: Convert the numbers into a DAG!
- Example 2: 3, 5, 1, 3, 11, 19, 4, 17, 21, 9, 13, 18
- Algorithm

Algorithm 0.2: LIS(A)

Binomial Coefficient

- $(x+y)^n = C(n,0)x^n + \cdots + C(n,k)x^{n-k}y^k + \cdots + C(n,n)y^n$
- Now, our problem is how to compute $C(n,k)$ for all $k = 0 \cdots n$ efficiently
- We know that $C(n,k) = \frac{n!}{k!(n-k)!}$, which is the combination size of picking k elements from n elements.
- Brute force algorithm: Compute $C(n,0), C(n,1), C(n,2), \cdots C(n,n)$ individually
- But we know that the same computations are repeated many times!
- In fact, we know that $C(n,k) = C(n-1,k-1) + C(n-1,k)$
- This idea has been discovered many many years ago in China, India, Iran, and Italy, etc, but one of its most famous names is Pascal's Triangle named after Blaise Pascal, a french mathematician

(image of Blaise Pascal 1623–1662)

Binomial Coefficient

- Example: $C(7,k)$, $k = 0, \cdots, 7$

	0	1	2	3	4	5	6	7
0								
1								
2								
3								
4								
5								
6								
7								

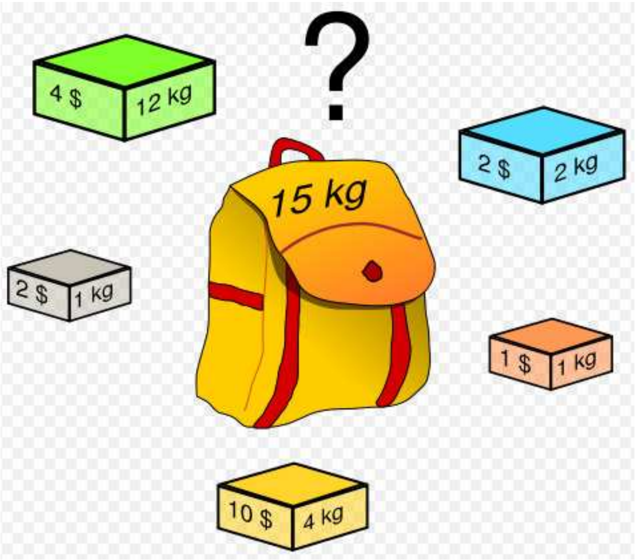
- Algorithm

Algorithm 0.3: BINOMIAL(n)

- Time complexity

Knapsack Problem

- Knapsack Problem: Given n objects, each object has weight w and value v, and a knapsack of capacity W, find most valuable items that fit into the knapsack



- Brute force approach
  - generate a list of all potential solutions
  - evaluate potential solutions one by one
  - when search ends, announce the solution(s) found
- What is the time complexity of the brute force algorithm?

Knapsack Problem

- Dynamic programming approach
  - Assume that we want to compute the optimal solution $S(w, i)$ for capacity $w < W$ with i items
  - Assume that we know the optimal solutions $S(w', i')$ for all $w' \leq w$ and $i' \leq i$
  - Option 1: **Don't add** the k-th item to the bag, then $S(w, i) = S(w, i-1)$
  - Option 2: **Add** the k-the item to the bag, then $S(w, i) = S(w - w_i, i-1) + v_i$

w	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12kg, \$4															
1kg, \$2															
2kg, \$2															
1kg, \$1															
4kg, \$10															

- Time complexity?

Knapsack Problem and Memory Functions

- So far, we look at four DP-based algorithms, all of them are bottom-up approaches.
- We can in fact design DP-based algorithms using top-down (recursive) approach.
  - One important benefit of top-down approach is that we can avoid solving unnecessary subproblems

w	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
12kg, $4															
1kg, $2															
2kg, $2															
1kg, $1															
4kg, $10															

- Algorithm

Algorithm 0.4: NAPSK(w, i)

if $V[w, i] < 0$
  then {
    if $w < W[i]$
      then $value \leftarrow$ NAPSK$(w, i-1)$
      else $w < W[i]$
      then $value \leftarrow \max\{$NAPSK$(w, i-1)$, NAPSK$(w - W[i], i-1) + V[i]\}$
    $V[w, i] \leftarrow value$
  }
return $(V[w, i])$

Summary

- Things you need to know about dynamic programming (dp)
  - **programming** in dp (and linear programming) is a mathematical term, which means **optimization** or planning, i.e. it should not be confused with "computer programming" or "programming language"
  - dp solves problems with **overlapping sub-problems**
  - dp solves problems which have **optimal substructure**, i.e., its optimal solution can be constructed from optimal solutions of its sub-problems
  - dp stores the results of sub-problems for later reuse
  - dp works by converting a problem into a set of sub-problems and representing these sub-problems as a DAG.
- Next week: Dynamic Programming 2
  - Edit distance (string matching)
  - Chain matrix multiplication
  - All pairs shortest distance