11/8/18

Dynamic Programming

→ **Knapsack**:

n items each with weight $w_i > 0$ (integer) and value $v_i > 0$.

Knapsack of capability W weight.

Goal: Find subset of items of largest total value that fits in knapsack.

Idea: $OPT(\dots) = \{$ value of the optimal solution.

$OPT(\dots)$ → smaller part / problem

- use memoisation / recursion.

? How do we decide what is a smaller problem / subprob

$1, 2, 3, \dots, n$

Subproblem: $1 \leq n$ looks $1, \dots, i$ } One possible subproblem.

Case 1: item i is not in the optimal solution

$OPT(i, w) = OPT(i-1, w) \Rightarrow OPT(i, w) = OPT(i-1, w)$

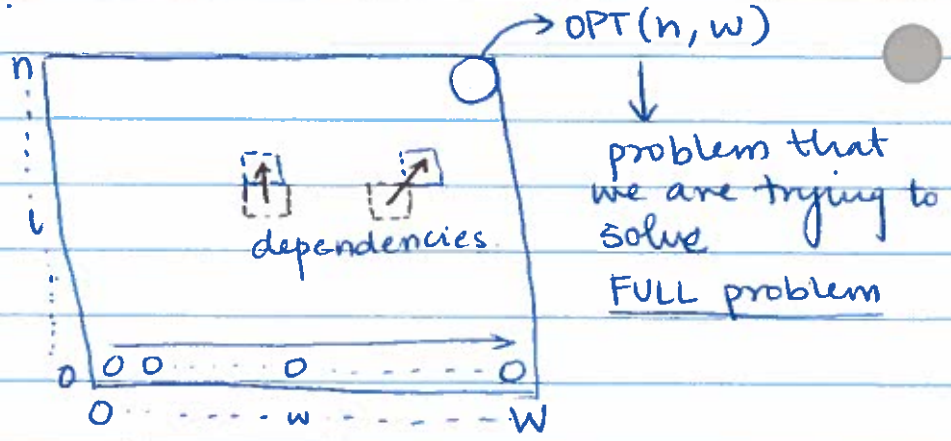
Case 2: item i is in optimal solution.

$OPT(i, w) = v_i + OPT(i-1, w-$

$$OPT(i,w) = \begin{cases} 0 & \text{if } i = 0 \\ OPT(i-1, w) & \text{if } w - w_i < 0 \\ \max\{OPT(i-1, w),\ v_i + OPT(i-1, w-w_i)\} & \text{Otherwise} \end{cases}$$

Can be negative so need to account for this

→ We need a 2-D array for our "Bottom-Up" iterative solution!



Why is it top-right corner?

→ 0 ... n on the vertical axis is not selecting n objects it represents the problem size.

Are the weights integers?

Yes, yes they are.

⇓ Refer to the slides for the iterative algorithm

→ Running Time: $\Theta(nW)$ → depends on capacity of knapsack

- Need to assume integer weights → limitation

? Can we make it independent of capacity of knapsack? → Efficient solutio np hard prob

- Another approach (Brute force algo) → enumerate all subsets of items and get one with max value within capacity of knapsack

$\Omega(2^n)$ → independent of capacity of knapsack.

→ Optimize memory usage by remembering a row at a time → $\Theta(W)$ memory space complexity.

→ We still need to know which items to choose because OPT(n, W) just gives max value.

Run another algorithm to decide which item to tal

[How to do this is on the homework!]

⟹ LONGEST COMMON SUBSEQUENCE:

Strings of Sequences:
- X = a b c b d a b
- Y = b d c a d a

Common subsequence ← CS :
- c b a (in this order in both X & Y)
- b d a b
- b c a b
- b c b a

} LCS (longest common subsequence)

→ Optimal Substructure properties: reduce big problem to smaller problems.

Let X, Y be two strings, let Z be a LCS of X, Y

X_m = first 'm' characters of X

Y_n = first 'n' characters of Y

Z_k = ~~first 'k' characters of Z~~ LCS of X_m and Y_n

(must be in LCS Z_k)

- if Last character of X_m = last char. of Y_n then Z_{k-1} is a LCS of X_{m-1} and Y_{n-1} (↳ remove last char.)

- if last char of X_m ≠ last char of Y_n → can't both be in LCS Z_k