Lecture 20: Dynamic Programming III: Text Justification, Knapsack, Pseudopolynomial Time

Lecture Overview

- Review
- Bottom-Up Implementation
- Parent Pointers
- Text Justification
- Knapsack
- Pseudopolynomial Time

Readings

CLRS 15

Review:

* DP is all about subproblems & guessing

* 5 easy steps:

(a) define subproblems: count ♯ subprobs.

(b) relate subproblem solutions, usually by guessing (part of the solution): count ♯ choices
IMPORTANT: check that subproblem solutions are related acyclically—recall the problem with the obvious shortest path recursion in the last lecture!

(c) recurse + memoize

(d) time = ♯ subprobs × time/subprob.
= ♯ subprobs × ♯ guesses/subpr. × overhead for combining solutions

(e) check if original problem = a subproblem or solvable from DP table ($\Longrightarrow$ extra time)

* for sequences, good subproblems are often prefixes OR suffixes OR substrings

Bottom-up implementation of DP (Repetition From Previous Lecture):

Alternative to recursion

- subproblem dependencies form DAG (see Figure 4)—if not, we need a better recursive formulation of the problem
- imagine topological sorting the dependency graph
- iterate through subproblems in that order
  $\Longrightarrow$ when solving a subproblem, have already solved all dependencies
- often: "solve smaller subproblems first"

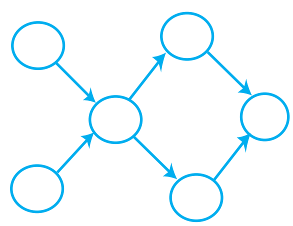

Figure 1: DAG.

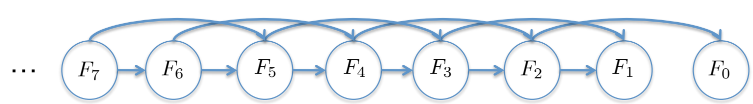

Figure 2: Subproblem Dependency Graph for Fibonacci Numbers.

Example.

Fibonacci:
for k in range($n+1$): fib[k] $= \cdots$

Shortest Paths:
for k in range(n): for v in V : $d[k, v, t] = \cdots$

Crazy Eights:
for i in reversed(range(n)): trick[i] $= \cdots$

Longest Common Subsequence:
c(i,j) = length of the LCS(x[i:],y[j:])
Recall Recursive formula:

$$c(i,j) = \begin{cases} 1 + c(i+1, j+1), & \text{if } x[i] = y[j] \\ \max\{c(i+1,j), c(i,j+1)\}, & \text{if } x[i] \neq y[j] \end{cases} \quad (1)$$

base cases: $c(\mid x \mid, j) = c(i, \mid y \mid) = \emptyset$

Figure 3 shows Bottom-Up Strategies for LCS.

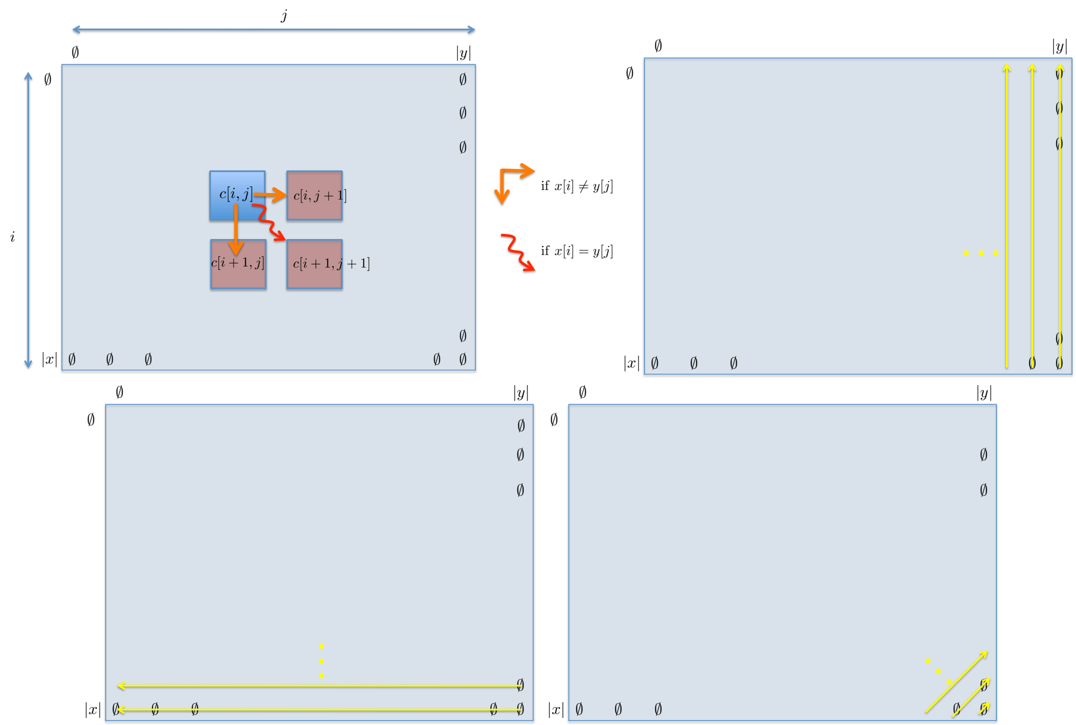

Figure 3: Subproblem Dependency Structure for Longest Common Subsequence, and different Bottom-Up Computation Strategies.

Parent Pointers

- Often straightforward DP returns the *value* of the optimal solution.
- To find the solution achieving this value, a bit more book-keeping is required.
- It is usually sufficient to remember for each subproblem what guess resulted in the optimal solution of the subproblem.
- E.g., in the LCS problem it is enough to remember, for all pairs i, j, the direction "right", "down", or "diagonal" achieving equality in (1).
- If we have these "pointers", we can just follow them starting at position $(0, 0)$ of the table to reconstruct the optimal solution.

Text Justification:

Split text into "good lines"

- obvious (MS Word/Open Office) algorithm: put as many words fit on first line, repeat
- but this can make *very bad* lines

	blah blah blah		blah blah
:(	b l a h	vs.	blah blah
	reallylongword		reallylongword

Figure 4: Good vs. Bad Justification

Mathematically the line justification problem:

- INPUT: Given array of words w[0 : n].
- SCORING RULE: Suppose we are considering a line ℓ containing the words $w[i]$ through $w[j]$. Define the <u>badness</u>(ℓ) for the line of words $\ell \equiv w[i : j+1]$ to be, e.g.,

$$\begin{cases} +\infty, & \text{if total\_length}(\ell) > \text{page\_width} \\ (\text{page\_width - total\_length}(\ell))^3, & \text{otherwise} \end{cases}$$

- GOAL: Split words into lines $\ell_1 = w[0 : i_1]$, $\ell_2 = w[i_1 : i_2]$, etc. to min $\sum_i$ badness(ℓ_i).

Subproblem structure:

1. subproblem DP[i]= min badness for suffix words $w[i :]$
   $\Longrightarrow$ $\sharp$ subproblems $= \Theta(n)$ where $n = \sharp$ words
2. guessing = where to end first line, in the optimal justification of words $w[i : n]$
   $\Longrightarrow$ $\sharp$ choices $= n - i + 1 = O(n)$
3. relation:
   - DP[i] = min(badness(i, j) + DP[j] for j in range($i + 1, n + 1$))
   - DP[n] = $\emptyset$
     $\Longrightarrow$ time per subproblem $= O(n)$
4. total time $= O(n^2)$
5. solution = DP[$\emptyset$]
   (& use parent pointers to recover split)

Knapsack:

Knapsack of size S you want to pack with a subset of n items,

- each item i has integer size s_i & real value v_i
- goal: choose subset of items of maximum total value subject to total size $\leq S$

Trivial Algorithm:

Try all possible subsets of the items $\Longrightarrow$ runtime exponential in the number of items.

First Attempt:

1. ~~subproblem $DP[i]$ = value for suffix [i:] of items~~ DOESN'T WORK, see below
2. guessing = whether to include item i $\Longrightarrow$ $\sharp$ choices = 2
3. relation:
   - $DP[i] = \max(DP[i+1], v_i + DP[i+1]$ if ~~$s_i \leq S$~~?!)
   - not enough information to know whether item i fits - how much space is left? GUESS!

Second Attempt, keeping more info:

1. subproblem $DP[i, X]$ = value for suffix $[i:]$ of items, given knapsack of size X
   $\Longrightarrow$ $\sharp$ subproblems $= O(nS)$!
2. guessing: whether to include i or not in the optimal knapsack of size X
3. relation:
   - $DP[i, X] = \max(DP[i+1, X], v_i + DP[i+1, X - s_i]$ if $s_i \leq X)$
   - $DP[n, X] = \emptyset$
     $\Longrightarrow$ time per subproblem $= O(1)$
4. total time $= O(nS)$
5. solution $= DP[\emptyset, S]$
   (& use parent pointers to recover subset)
   AMAZING: effectively trying all possible subsets!

Knapsack is in fact NP-complete! $\Longrightarrow$ suspect no polynomial-time [1] algorithm (polynomial in length of input).

[1] More on NP-completeness later in the term. For now, NP-complete problems is a family of hard problems for which no polynomial-time algorithm is known.

Why isn't the above algorithm polynomial time?

- here input $=< S, s_0, \cdots, s_{n-1}, v_0, \cdots, v_{n-1} >$
- length in binary: $O(\lg S + \lg s_0 + \cdots) \approx O(n \lg \ldots)$
- so $O(nS)$ is not "polynomial-time", because S is exponential in $\log S$, an it could be that $\log S$ dominates the size of the input
- $O(nS)$ still pretty good if S is small
- "pseudopolynomial time": polynomial in length of input & integers in the input

Remember:
polynomial - GOOD
exponential - BAD
pseudopoly - SO SO