6.006- Introduction to Algorithms

Lecture 20

Prof. Patrick Jaillet
Lecture overview

Dynamic Programming III

- review: longest common subsequence (LCS)
- recursion + memoization v.s. bottom up
  - (illustration with LCS)
- use of parent pointers
  - (illustration with LCS)
- knapsack problem
- text justification
Longest Common Subsequence (LCS)

- given two sequences $x[1..m]$ and $y[1..n]$, find a longest subsequence $LCS(x,y)$ common to both:

$x$: A B C B D A B

$y$: B D C A B A

- denote the length of a sequence $s$ by $|s|
- first get $|LCS(x,y)|$
LCS: A recurrence

• consider prefixes of x and y
  – x[1..i] ith prefix of x[1..m]
  – y[1..j] jth prefix of y[1..n]
• define $c[i,j] = |\text{LCS}(x[1..i],y[1..j])|

$$
c[i,j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0, \\
 c[i-1,j-1] + 1 & \text{if } i,j > 0 \text{ and } x_i = y_j, \\
 \max(c[i-1,j],c[i,j-1]) & \text{if } i,j > 0 \text{ and } x_i \neq y_j.
\end{cases}
$$

running time is .... $O(n \times m)$.... (if done well !)
LCS recursion + memoization

\[
c[\alpha, \beta] = \begin{cases} 
0 & \text{if } \alpha \text{ empty or } \beta \text{ empty,} \\
\ c[\text{prefix } \alpha, \text{prefix } \beta] + 1 & \text{if end}(\alpha) = \text{end}(\beta), \\
\ \max(c[\text{prefix } \alpha, \beta], c[\alpha, \text{prefix } \beta]) & \text{if end}(\alpha) \neq \text{end}(\beta). 
\end{cases}
\]
LCS – bottom up & pointers

```plaintext
|LCS(x, y)|
m ← length[x]
n ← length[y]
for i ← 1 to m
do c[i, 0] ← 0
for j ← 0 to n
do c[0, j] ← 0
for i ← 1 to m
  do for j ← 1 to n
do if x_i = y_j
  then c[i, j] ← c[i-1, j-1] + 1
      p[i, j] ← “↖”
  else if c[i-1, j] ≥ c[i, j-1]
    then c[i, j] ← c[i-1, j]
       p[i, j] ← “↑”
  else c[i, j] ← c[i, j-1]
      p[i, j] ← “←”
return c and p
```

\[
c[i, j] = \begin{cases} 
0 & \text{if } i = 0 \text{ or } j = 0, \\
(c[i-1, j-1] + 1 & \text{if } i, j > 0 \text{ and } x_i = y_j, \\
\max(c[i-1, j], c[i, j-1]) & \text{if } i, j > 0 \text{ and } x_i \neq y_j.
\end{cases}
\]
Example

$x$:  A  B  C  B

$y$:  B  D  C

<table>
<thead>
<tr>
<th></th>
<th>$y_j$</th>
<th>B</th>
<th>D</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>$x_i$</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>↑0</td>
<td>↑0</td>
<td>↑0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>↑1</td>
<td>↑1</td>
<td>↑1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>↑1</td>
<td>↑1</td>
<td>↑2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>↑1</td>
<td>↑2</td>
<td></td>
</tr>
</tbody>
</table>
Use of parent pointers

• we found length of LCS, what about actual LCS?
• using the “parent pointers” p
  – p remembers if c[i,j] used c[i-1, j-1], c[i, j-1], or c[i-1,j]
  – starting at c[m,n]:
    • if c[m-1,n-1], then x[m]=y[n] is part of opt
      – put it at end and output opt from c[m-1,n-1]
    • else, output opt from c[m-1,n] or c[m,n-1]
Constructing an LCS

\[
\text{PRINT-LCS} \ (p, \ x, \ i, \ j) \\
\text{if} \ i = 0 \text{ or } j = 0 \ \\
\quad \text{then return} \\
\text{if } p[i, j] = "\" \\
\quad \text{then PRINT-LCS}(p, \ x, \ i-1, \ j-1) \\
\quad \text{print } x_i \\
\quad \text{elseif } p[i, j] = "↑" \\
\quad \text{then PRINT-LCS}(p, \ x, \ i-1, \ j) \\
\text{else PRINT-LCS}(p, \ x, \ i, \ j-1)
\]

initial call is PRINT-LCS \ (p, \ x, \ m, \ n) \\
running time: \ \mathcal{O}(m+n)
Example

\[ x: \quad A \quad B \quad C \quad B \]

\[ y: \quad B \quad D \quad C \]

<table>
<thead>
<tr>
<th></th>
<th>( y_j )</th>
<th>B</th>
<th>D</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>( x_i )</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>A</td>
<td>0</td>
<td>↑0</td>
<td>↑0</td>
<td>↑0</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>↑1</td>
<td>↑1</td>
<td>↑1</td>
</tr>
<tr>
<td>C</td>
<td>0</td>
<td>↑1</td>
<td></td>
<td>↑2</td>
</tr>
<tr>
<td>B</td>
<td>0</td>
<td>↑1</td>
<td>↑1</td>
<td>↑2</td>
</tr>
</tbody>
</table>
Bottom-Up DP

• we’ve been looking at DP recurrences
  – which suggests recursive implementations
  – and memoize results as you get them
• can also solve “bottom up”
  – compute sub-problems before super-problem
  – put results in memo table for later use
• how to order problems to ensure this works?
The DP DAG

• define a graph representing DP
  – sub-problems are vertices
  – edge $x \rightarrow y$ if problem $x$ depends on problem $y$
• what order of problem solving works?
  – need order where $x$ follows $y$ if $x \rightarrow y$
  – Topological Sort!
  – can do so if graph is a DAG
  – what if not?
    • cyclic problem dependency
    • can’t use DP
Knapsack Problem

- Knapsack (or cart) of size $S$
- Collection of $n$ items; item $i$ has size $s_i$ and value $v_i$
- Goal: choose subset with $\sum_i s_i < S$ maximizing $\sum_i v_i$
- Ideas?
  - try all possible subsets: $2^n$
  - greedy?
    - choose items maximizing value?
    - choose items maximizing value/size
      - what if they don’t exactly fit?
Some bad and better news

• For arbitrary (real), Knapsack is hard (NP-hard)
  – no polynomial time algorithm in 30 years of trying
  – it’s exactly as hard as several thousand other important problems
  – and we haven’t been able to find polynomial time algorithms for them for 30 years of trying either
  – most folks think there is none
• Better news:
  – There is a DP algorithm if sizes are integers
First attempt

• subproblem?
  – $\text{Val}[i] = $ Best value obtained for items[i:n]

• guess?
  – whether or not to include item i

• recurrence?
  – $\text{Val}[i] = \text{Val}[i+1]$
    or $v_i + \text{Val}[i+1]$ if total size < $S$?

• not a well-defined recurrence: doesn’t have enough info to tell if item i will fit
Second Attempt

• Solve a more complicated problem
  – initial problem is a special case
  – the complicated version has a recursion

• Val[i,X] = max value for items[i:n] if space is X

• Recurrence:
  – if $s_i > X$ then don’t include $i$, otherwise decide with
  – $Val[i, X] = \max(Val[i + 1, X], v_i + Val[i + 1, X - s_i])$
  – Opt = Val[0,S]
Analysis

• Is the recurrence a DAG?
  – yes, each problem depends on bigger i and smaller X
  – compute by decreasing i and increasing X
• Runtime?
  – each subproblem has 2 guesses: O(1)
  – one subproblem for each i, X<S
  – O(nS) subproblems
  – Total time: O(nS)
• Is this polynomial?
Text Justification – Word Processing

• A user writes stream of text
• WP has to break it into lines that aren’t too long
• obvious algorithm => greedy:
  – put as much on first line as possible
  – then continue to lay out rest
  – used by MSWord, OpenOffice
• Problem: suboptimal layouts !!
A Better Approach

• define an objective function
  – measure of how good a given layout is
  – not an algorithm, just a metric

• optimize the objective
  – here’s where you think of algorithm
Layout Function

• want to penalize big spaces
• what objective would do that?
  – sum of leftover spaces?
  – that’s constant for a given number of lines (just total space minus number of characters)
• should penalize big spaces “extra”
  – (LaTeX uses sum of cubes of leftovers)
Formalize

• input: array of words (lengths) w[0..n]
• split into lines L_1, L_2 ...
• badness(L) = (page width – total length(L))^3
  – (or ∞ if total length > page width)
• objective: break into lines L_1, L_2… minimizing \( \sum_i \text{badness}(L_i) \)
Can We DP?

- Subproblems?
  - \( DP[i] = \min \text{badness for words } w[i:n] \)
  - \( n \) subproblems where \( n \) is number of words
- Guesses for problem \( i \)?
  - Where to end first line in optimal layout
- Recurrence?
  - \( DP[i] = \min \text{badness}(i,j) + DP[j] \) for \( j \) in range\((i+1,n)\)
  - \( DP[n]=0 \)
  - \( \text{OPT} = DP[0] \)
- Runtime? \( O(n^2) \)?