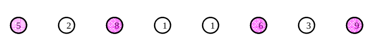
Lecture 16, Nov 20 2014

Dynamic Programming

- Main problem with greedy approaches:
  sometimes we cannot commit up-front.
- Dynamic programming:
  - » Meta-technique, not a specific algorithm.
- Main idea:
  - » solve many small sub-problems,
  - » combine solution to several small sub-problems to solve larger sub-problems.
  - » continue combining until we solve the original problem.
- Contrast with greedy

Simple example

- Problem:
  - » Given a line with nodes $v_1, v_2, \ldots, v_n$.



  - » Each node v_i has positive weight w_i
  - » Choose a subset of nodes such that
    - – Total weight of the chosen nodes is maximized
    - – No "neighbors" chosen, i.e. if v_i chosen, then neither v_{i+1} nor v_{i-1} chosen
    - – For simplicity: output only the value of maximum weight set (do not list nodes)
- What is the problem with greedy approach ?

Simple example - continued

- Subproblems:

  W_i = maximum weight if we only consider nodes **1** through i.
  W_0 =**0**

- Observation: $W_{i+1} = \max\{W_i \ , \ W_{i-1}+w_{i+1}\}$
  Either
  - » do not use w_{i+1}
  - » use w_{i+1} , in which case do not use w_i

- Running time ?
  - » Constant time per iteration
  - » n iterations total
  - » $\Theta(n)$

Longest Common Subsequence

- Consider two sequences:
  x = A B C B D A B |x| = m
  y = B D C A B A |y| = n

- Subsequence – "keep order, delete some elements"

- Goal: Find longest subsequence common to x and y

- Greedy: does not seem to work

- Brute force:
  - » Consider all substrings of x, all substrings of y, compare.
    Total: $O(2^{m+n}(m+n))$.
  - » Better: take any substring of x, check against y.
    Total: $O(2^m n)$, still too slow !

Main step

A B
B D C

A B C
B D C

A B C
B D C

191

Optimum Substructure

A **B** **C** **B** D **A** B
B D **C** A **B** **A**

- Define subproblems: $C(i,j)=LCS(x_1,x_2,\ldots,x_i,y_1,y_2,\ldots,y_j)$
- Observe that C(m,n) is the answer that we seek.
- Theorem:

$$C(i,j)=\begin{cases} C(i-1,j-1)+1 & \text{if } x_i=y_j \\ \max\{C(i,j-1),C(i-1,j)\} & \text{otherwise} \end{cases}$$

- Proof: Case 1, $x_i=y_j$
  - Consider $Z=z_1,\ldots z_k$ LCS of $(x_1\ldots x_i)$, $(y_1\ldots y_j)$
  - if $z_k \neq x_i$, then z is not LCS !!! (Why ??)
  - Now we claim that $z_1\ldots z_{k-1}$ is LCS of $(x_1\ldots x_{i-1})$, $(y_1\ldots y_{j-1})$
  - Proof: if there is a longer than Z-z_k = $z_1\ldots z_{k-1}$ sequence, just extend it and claim contradiction

192

Proof: continued

$$C(i,j)=\begin{cases} C(i-1,j-1)+1 & \text{if } x_i = y_j \\ \max\{C(i,j-1), C(i-1,j)\} & \text{otherwise} \end{cases}$$

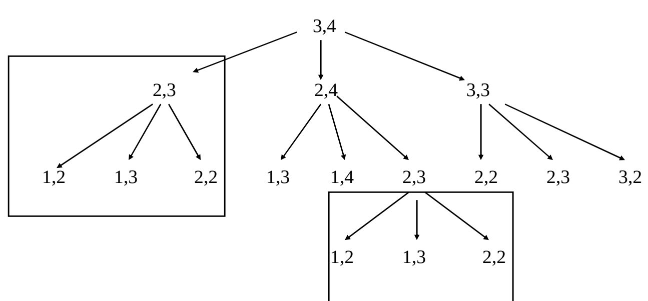
- Case 2: $x_i \neq y_j$
  either: $z_k = x_i$ (2a)
  or $z_k = y_j$ (2b)
  or not equal to either of them. (2c)
  - Case 2a: $z_k = x_i \Longrightarrow z_k \neq y_j$
    $\Longrightarrow z_1, \ldots z_k$ is a LCS of $(x_1 \ldots x_i)$, $(y_1 \ldots y_{j-1})$ (Why ??)
  - Case 2b is symmetric.
  - Do Case 2c at home.

193

Recursive algorithm

$$C(i,j)=\begin{cases} C(i-1,j-1)+1 & \text{if } x_i = y_j \\ \max\{C(i,j-1), C(i-1,j)\} & \text{otherwise} \end{cases}$$

- We can use the theorem to construct a recursive algorithm.
- Depth of the tree is $O(m+n)$, leads to $O(3^{m+n})$ bound, too large !



- Repeating sub-questions !

194

Analysis

- Main idea: we see repeating sub-questions
- How many different sub-questions ?
  - » One per $C(i,j)$ calculation: only $O(mn)$ different ones !
- Two different ways to implement:
  - » memoization: after computing sub-problem answer, remember it.
  - » dynamic programming: compute the table bottom-up.

$$C(i,j) = \begin{cases} C(i-1, j-1)+1 & \text{if } x_i = y_j \\ \max\{C(i, j-1), C(i-1, j)\} & \text{otherwise} \end{cases}$$

Computing the table

- Fill the table starting from top-left corner, and going row-by-row:

	y_i	B	D	C	A	B	A
x_i							
		0	0	0	0	0	0
A	0	0	0	0	1	1	1
B	0	1	1	1	1	2	2
C	0	1	1	2	2	2	2
B	0	1	1	2	2	3	3
D	0	1	2	2	2	3	3
A	0	1	2	2	3	3	4
B	0	1	2	2	3	4	4

- Each element depends on the one above, one left, and if x_i=y_i , then it is one more then the diagonal up-left element.
- Think about dynamic programming as "filling the table of subproblems"
- Possible implementation to allow trace-back: add flag [up | down | diagonal] to each entry

Matrix chain multiplication

- Consider the following chain: $A_1 \times A_2 \times \cdots \times A_n$, A_1 is $[p_o \times p_1]$, A_2 is $[p_1 \times p_2]$, etc.

  $[A_1 \times A_2]_{i,j} = \sum_{k=1}^{p_1} A_1[i,k] A_2[k,j]$, time $\approx$ $p_0 p_1 p_2$

- Example: [5x100] [100x2] [2x50]
  - Multiplying last two and then by the first one: 100x2x50 + 5x100x50 = 35,000 multiplications.
  - Multiplying first two and then by the last one: 5x100x2 + 5x2x50 = 1500

- *Order of multiplication affects the amount of work !*

Solving matrix chain multiplication

- Why can't we just use as subproblems the time to multiply matrices 1 to i??

- Observation:
  - Consider last optimum multiplication: $(A_1 \times \cdots \times A_k) \times (A_{k+1} \times \cdots \times A_n)$
  - Then both $(A_1 \times \cdots \times A_k)$ and $(A_{k+1} \times \cdots \times A_n)$ were computed optimally !! (Why ??)

- Subproblems: $m(i,j)$ is best "time" to multiply $(A_i \times \cdots \times A_j)$

$$m(i,j) = \begin{cases} 0 & \text{if } i = j \\ \min_{i \le k < j} \{ m(i,k) + m(k+1,j) + p_{i-1} p_k p_j \} & \text{if } i < j \end{cases}$$

- Answer is $m(1,n)$

Matrix chain continued

- Lets try to analyze using recurrence relation: $m(i,j)=\begin{cases} 0 & \text{if } i=j \\ \min_{i\le k<j}\{m(i,k)+m(k+1,j)+p_{i-1}p_k p_j\} & \text{if } i<j \end{cases}$

  $T(1)\ge 1$

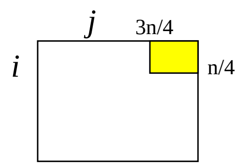
  $T(n)\ge 1+\sum_{k=1}^{n-1}[T(k)+T(n-k)+1]\ge 2\sum_{k=1}^{n-1}T(k)+n$

  by induction, easy to show that $T(n)\ge 2^{n-1}$

- Wrong approach ! There are only $O(n^2)$ different subproblems !

- Build the table bottom up, for decreasing row index i.
  Alternatively, start with diagonal, then elements $m(i,i+1)$, then elements $m(i,i+2)$, etc.

- $O(n)$ per each $m(i,j)$, total $O(n^3)$. Why is it $\Omega(n^3)$?



Computing $m(i,j)$ takes about j-i time
j-i is at least n/2 in yellow area
size of area n/4 by n/4

199