ECE 235: Lec 7: Dynamic Programming 10/16/01 (Ch. 15)

Pai H. Chou, University of California at Irvine

Homework problems (Due Tue. Oct 23)

- 15.2-1 (page 338), 15-7 (page 369)
- 16.1-3 (page 379)
  item Python programming: implement the Huffman code algorithm in Section 16.3 (page 388) and test it with (a) example in Figure 16.5, plus (b) one test case that you create.

This lecture: Dynamic programming

- Optimization problems and Dynamic programming
- Assembly line scheduling
- matrix multiply
- longest common subsequencce
- optimal binary search tree

1 Optimization problems

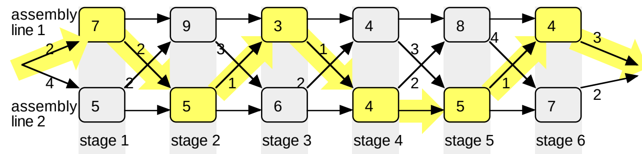
- many possible solutions with different costs
- want to maximize or minize some cost function
- unlike sorting – it's sorted or not sorted.. partially sorted doesn't quite count.
- examples: matrix-chain multiply (same results, just faster or slower)
  knap-sack problem (thief filling up a sack), compression

1.1 Dynamic programming

- "programming" here means "tabular method"
- Instead of re-computing the same subproblem, save results in a table and look up (in constant-time!)
- significance: convert an otherwise exponential/factorial-time problem to a polynomial-time one!
- Problem characteristic: *Recursively decomposable*
  - Search space: a lot of repeated sub-configurations
  - optimal substructure in solution

2 Case study: Assembly line scheduling

- two assembly lines 1 and 2
- each line i has n stations $S_{i,j}$ for n stages of assembly process
- each station takes time $a_{i,j}$
- chassis at stage j must travel stage $(j+1)$ next
  - option to stay in same assembly line or switch to the other assembly line
  - time overhead of $t_{i,j}$ if decided to switch to line to go to $S_{i,j}$.



- Want to minimize time for assembly
  - Line-1 only: $2+7+9+3+4+8+4+3=40$
  - Line-2 only: $4+8+5+6+4+5+7+2=41$
  - Optimal: $2+7+(2)+5+(1)+3+(1)+4+5+(1)+4+3=38$
- How many possible paths? 2^n (two choices each stage)

Optimal substructure

- Global optimal contains optimal solutions to subproblems
- Fastest way through any station $S_{i,j}$ must consist of
  - shortest path from beginning to $S_{i,j}$
  - shortest path from $S_{i,j}$ to the end
  - That is, cannot take a longer path to $S_{i,j}$ and make up for it in stage $(j+1)\ldots n$.
- Notation: $f_i[j]$ = fastest possible time from start through station $S_{i,j}$ (but not continue)
  e_i, x_i are entry/exit costs on line i
  Goal is to find f^* global optimal
- initially, at stage 1 (for line $l=1$ or 2),
  $f_l[1]=e_l(\text{entry time})+a_{l,1}$ (assembly time)
- at any stage $j>1$, line l, (and m donotes "the other line")

$$\begin{aligned} f_l[j] &= \min\left\{ \begin{array}{ll} f_l[j-1] & \text{same line } l \\ f_m[j-1]+t_{m,(j-1)} & \text{other line } m \text{ + transfer} \end{array} \right\} \\ &+ a_{l,j} \text{ (assembly time at station } S_{l,j}) \end{aligned}$$

Can write this as a recursive program:

$\text{F}(i,j)$
  if $j=1$
    then return $e_i+a_{i,1}$
    else return $\min(\text{F}(i,j-1),$
          $\text{F}(i\%2+1,j-1)+t_{i\%2+1,j-1})$
        $+a_{i,j}$

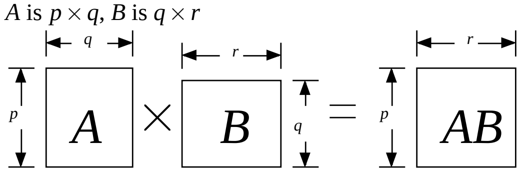
But! There are several problems:

- many repeat evaluation of F(i,j), could be $O(2^n)$ time
  $\Rightarrow$ use a 2-D array $f[i,j]$ to remember the running minimum
- does not track the path
  $\Rightarrow$ use array $l_i[j]$ to remember which path gave us this min
- Iterative version shown in book on p. 329.
- Run time is $\Theta(n)$.

3 Matrix-Chain multiply

Basic Matrix Multiply

- A is $p \times q$, B is $q \times r$



- product is $p \times r$ matrix: $c_{i,j} = \sum_{y=1\ldots q} a_{i,y} \cdot b_{y,j}$
- total number of scalar multiplications $= p \times q \times r$

Multiply multiple matrices

- matrix multiplication is associative:
  $\Rightarrow (AB)C = A(BC)$
  Both yield $p \times s$ matrix
- Total # multiplications can be different! (*added*)
- $(AB)C$ is

	pqr	to multiply AB first,
$+$	prs	to multiply $(AB)(p \times r)$ w/ $C(r \times s)$
$=$	$\boxed{pqr + prs}$	total # multiplications

- On the other hand, $A(BC)$ is $\boxed{pqs + qrs}$

Example: if $p = 10, q = 100, r = 5, s = 50$, then

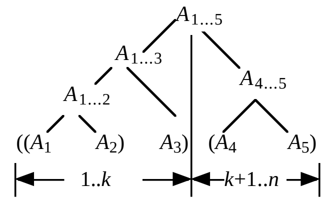
- $pqr + prs = 5000 + 2500 = 7500$
- $pqs + qrs = 50000 + 25000 = 75000 \Rightarrow$ ten times as many!

Generalize to matrix chain: $A_1, A_2, A_3 \ldots A_n$
But there are many ways!

$P(1) = 1$	(A)	$\triangleright$ *nothing to multiply*
$P(2) = 1$	(AB)	
$P(3) = 2$	$A(BC), (AB)C$	
$P(4) = 5$	$A(B(CD)), A((BC)D), (AB)(CD), (A(BC))D, ((AB)C)D$	
$P(5) = 14$	$\ldots \Rightarrow$ Exponential growth	

$$P(n) = \begin{cases} 1 & \text{if } n = 1, \\ \sum_{k=1}^{n-1} P(k) \times P(n-k) & \text{for } n \geq 2 \\ \Omega(4^n / n^{3/2}) & \text{at least exponential!} \end{cases}$$

3.1 optimial parenthesization to inimize # scalar mult's



Notation:

- let $A_{i\ldots j}$ denote matrix product $A_i A_{i+1} \cdots A_j$
- matrix A_i has dimension $p_{i-1} \times p_i$

Optimal substructure

- if optimal parenthesization for $A_{1\ldots j}$ at the top level is $(L)(R) = (A_{1\ldots k})(A_{k+1\ldots j})$, then
  - L must be optimal for $A_{1\ldots k}$, and
  - R must be optimal for $A_{k+1\ldots j}$
- Proof by contradiction

Let $M(i,j)$ = Minimum cost from the i^{th} to the j^{th} matrix

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

As a recursive algorithm (very inefficient!):

M(i,j)
  if $i = j$
    then return 0
    else return M(i,k)+ M$(k+1,j) + p_{i-1} p_k p_j$

Observation

- don't enumerate the space!
  bottom up $\Rightarrow$ no need to take the min so many times!
- instead of recomputing $M(i,k)$, remember it in array $m[i,k]$
- book keeping to track optimal partitioning point. See Fig.1
- $O(n^3)$ time, $\Theta(n^2)$ space (for m and for s arrays)

4 Longest common subsequence

- Example sequence $X = \langle$`A, B, C, B, D, A, B`$\rangle$,
  $Y = \langle$`B, D, C, A, B, A`$\rangle$
- a subsequence of X is $Z = \langle$`B, C, D, B`$\rangle$
- Longest common subsequence (LCS) of length 4:
  $\langle$`B, C, B, A`$\rangle$, $\langle$`B, D, A, B`$\rangle$
- This is a maximization, also over addition, but add cost by 1 (length increment)

```
MATRIX-CHAIN-ORDER(p)
n ← length[p] − 1
for i ← 1 to n
    do m[i,i] ← 0
for l ← 2 to n:     ▷ l = length of interval considered
    do for i ← 1 to (n − l) + 1:
        ▷ starting index, from 1 up to n− length for each length
        do j ← i + l − 1
            ▷ ending index, always length away from the starting index
            m[i, j] ← ∞
            for k ← i to j − 1:
                ▷ different partitions between i and j
                do q ← m[i,k] + m[k+1, j] + p_{i−1} p_k p_j.
                    if (q < m[i, j]):
                        then m[i, j] ← q
                            s[i, j] ← k ▷ remember best k between i, j
return m, s
```

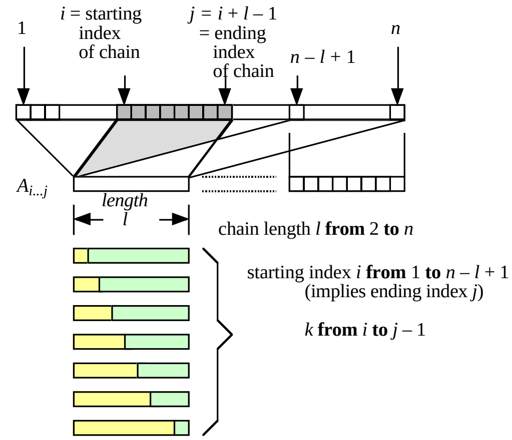


Figure 1: Matrix-Chain-Order Algorithm and graphical illustration.

Brute force:

- enumerate all subsequences of x (length m), check if it's a subsequence of y (length n)
- #of subsequences of $x = 2^m$ (binary decision at each point whether to include each letter)
- worst case time is $\Theta(n \cdot 2^m)$ because for each one check against y's length $= n$

Better way:

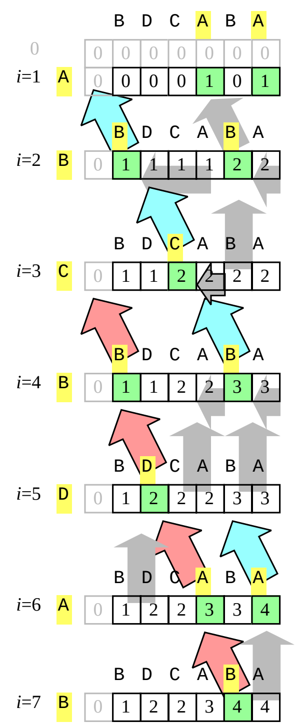
- Notation: X_k = length-k prefix of string X
  x_i is i^{th} character in string X
- $Z = \langle z_1 \dots z_k \rangle$ is an LCS of $X = \langle x_1 \dots x_m \rangle$ and $Y = \langle y_1 \dots y_n \rangle$
- if $x_m = y_n$ then $z_k = x_m = y_n$, and
  Z_{k-1} is an LCS of X_{m-1}, Y_{n-1}.
- if $x_m \neq y_n$ then
  - if $z_k \neq x_m$ then Z is LCS of X_{m-1}, Y
  - if $z_k \neq y_n$ then Z is LCS of X, Y_{n-1}.
- $c[i, j]$ = LCS length of X_i, Y_j

$$c[i,j] \quad = \quad \begin{cases} c[i-1, j-1] + 1 & \text{if } x[i] = x[j] \text{(match)} \\ \max \left\{ \begin{array}{l} c[i, j-1] \\ c[i-1, j] \end{array} \right\} & \text{(no match, advance either)} \end{cases}$$

Algorithm

```
c[1 : m, 0] ← 0
c[0, 1 : n] ← 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
                b[i, j] ← "match" (↖)
            else if (c[i − 1, j] ≥ c[i, j − 1])
                then c[i, j] ← c[i − 1, j] ▷ copy the longer length
                    b[i, j] ← "dec i" (↑)
                else ▷ c[i, j − 1] > c[i − 1, j]
                    b[i, j] ← "dec j" (←)
```

i \ j		0	1	2	3	4	5	n =6
			B	D	C	A	B	A
0		0	0	0	0	0	0	0
1	A	0						
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
m=7	B	0						



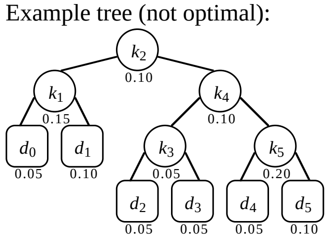
Time $\Theta(mn)$, Space $\Theta(mn)$.

5 Optimal Binary Search Trees

- input: n keys $K = \langle k_1, k_2, \dots, k_n \rangle$
  $n+1$ dummy keys $D = \langle d_0, d_1, \dots, d_n \rangle$
- $d_0 < k_1 < d_1 < k_2 < d_2 < \dots < k_n < d_n$
- key k_i has probability p_i, and
  dummy key d_i has probability q_i, and

$$\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$$

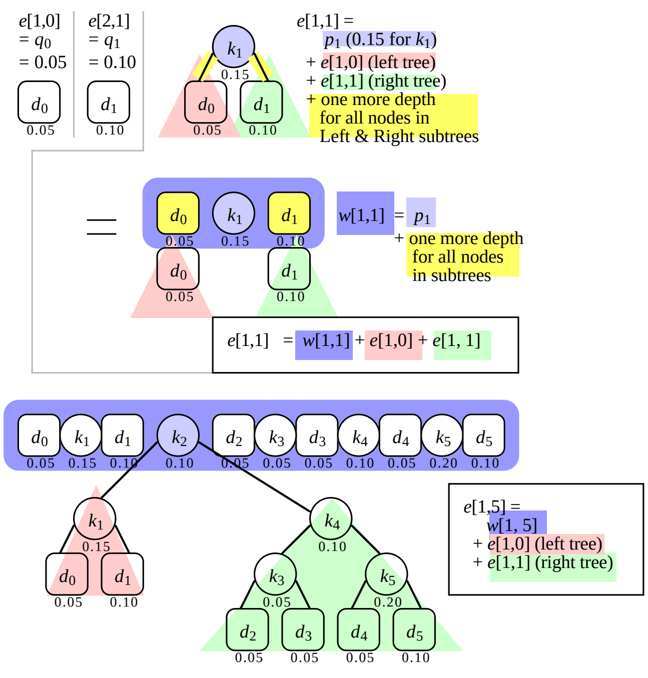
- want: Binary tree that yields fastest search: (fewer steps) for frequently used words
- k_i keys should be internal nodes, and d_i dummy keys should be leaves.
- optimize for common case. Balanced tree might not be good!
- Example tree (not optimal):



Expected search cost of tree T

- Optimal substructure: if root=k_r, $L = (i \ldots r-1)$, $R = (r+1 \ldots j) \Rightarrow L, R$ must be optimal subtrees.
- Expected cost $e[i, j]$

$$e[i,j] = \begin{cases} q_{i-1} & (\textit{a dummy leaf}) \quad \text{if } j = i-1 \\ \min_{i \le r \le j}\{ \quad e[i, r-1] & (\textit{left subtree}) \\ \qquad +e[r+1, j] & (\textit{right subtree}) \\ \qquad +w(i,j) & (\textit{add one depth}) \quad \}\text{if } i \le j \end{cases}$$



- use arrays to remember $e[i, j], w[i, j]$ instead of recomputing
- use array $root[i, j]$ to remember root positions

OPTIMAL-BST(p, q, n) (page 361)

```
for i ← 1 to n+1
    do e[i, i−1] ← q_{i−1}
       w[i, i−1] ← q_{i−1}
for l ← 1 to n
    do for i ← 1 to n−l+1
        do j ← i+l−1
           e[i, j] ← ∞
           w[i, j] ← w[i, j−1] + p_j + q_j
           for r ← i to j
               do t ← e[i, r−1] + e[r+1, j] + w[i, j]
                  if t < e[i, j]
                     then e[i, j] ← t
                          root[i, j] ← r
return e, root
```