Lakehead UNIVERSITY

COMP 4433: Algorithm Design and Analysis

Dr. Y. Gu

Feb. 1, 2023 (Lecture 6)

Dynamic Programming (More Examples)

Example 3:

Matrix-Chain Multiplication Problem

Matrix-Chain Multiplication

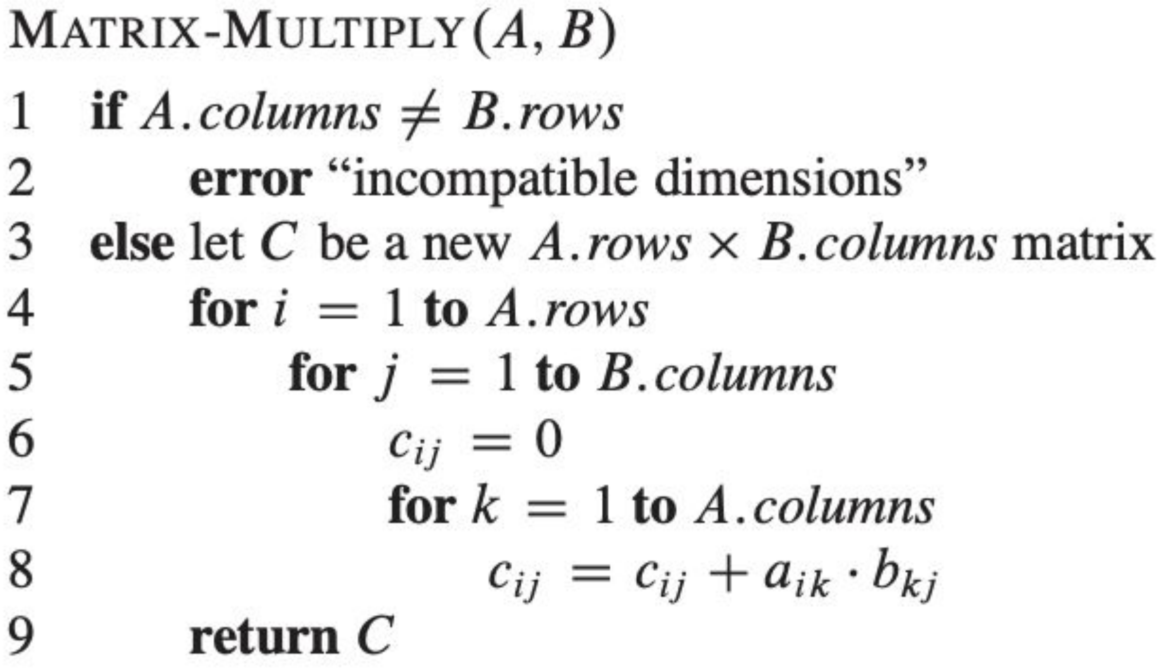
- Suppose n matrices A_1, A_2 ,⋯, A_n are given, where the matrices are not necessary square. We need to compute the product $A_1A_2 \cdots A_n$.
- Matrix multiplication is associative, and so all parenthesizations yield the same product.
- A product of matrices is fully parenthesized if it is either a single matrix or the product of two fully parenthesized matrix products, surrounded by parentheses.
- For example, if the chain of matrices is $<A_1, A_2, A_3, A_4>$, then we can fully parenthesize the product $A_1A_2A_3A_4$ in five distinct ways:

  $(A_1(A_2(A_3A_4)))$, $(A_1((A_2A_3)A_4))$, $(A_1A_2)(A_3A_4)$,

  $((A_1(A_2A_3))A_4)$, $(((A_1A_2)A_3))A_4)$

Why Care about Parenthesization?

- Consider first the cost of multiplying two matrices.



```
MATRIX-MULTIPLY(A, B)
1  if $A.columns \neq B.rows$
2      error "incompatible dimensions"
3  else let $C$ be a new $A.rows \times B.columns$ matrix
4      for $i = 1$ to $A.rows$
5          for $j = 1$ to $B.columns$
6              $c_{ij} = 0$
7              for $k = 1$ to $A.columns$
8                  $c_{ij} = c_{ij} + a_{ik} \cdot b_{kj}$
9      return $C$
```

The main cost of the procedure is from line 4 to line 8, which is A.rows × B.columns × A.columns scalar multiplications.

Why Care about Parenthesization?

- When we compute the multiplication of more than two matrices, the order of the multiplication will affect the cost.
- Consider the multiplication $A_1A_2A_3$. Suppose the dimensions of A_1,A_2,A_3 are 10×100,100×5,5×50, respectively.
  - If we multiply according to $((A_1A_2)A_3)$, then we first perform $10 \cdot 100 \cdot 5 = 5000$ scalar multiplications to compute a 10×5 matrix. Then multiply the resulting matrix with A_3, which needs $10 \cdot 5 \cdot 50 = 2500$ scalar multiplications. In this way, we need total 7500 multiplications.
  - But if we compute the multiply as $(A_1(A_2A_3))$, then a simple calculation shows that we need a total 75000 scalar multiplications.

Matrix-chain Multiplication Problem

Given a chain $\langle A_1, A_2, ..., A_n \rangle$ of n matrices, where for i = 1,2,...,n, matrix A_i has dimension $p_{i-1} \times p_i$, fully parenthesize the product $A_1 A_2 \ldots A_n$ in a way that minimizes the number of scalar multiplications.

This problem is to find out an optimal order of products for a matrix-chain multiplications, not actually multiplying matrices.

Number of Possible Parenthesizations

Denote the number of alternative parenthesizations of a sequence of n matrices by P(n). Then P(1) = 1. When n ≥ 2, a fully parenthesized matrix product is the product of two fully parenthesized matrix subproducts, and the split between the two subproducts may occur between the kth and (k + 1)st matrices for any k = 1, 2, . . . , n − 1. Thus we have

$$P(n) = \begin{cases} 1 & \text{if } n = 1\,, \\ \displaystyle\sum_{k=1}^{n-1} P(k)P(n-k) & \text{if } n \geq 2\,. \end{cases}$$

Number of Possible Parenthesizations

Using the substitution method, we can show that the solution to the recurrence in the previous slide is $\Omega(2^n)$. So an exhaustive search will be exponential in n.

We can see that in the recurrence, many value of P(k) is computed repeatedly. Therefore we can apply the dynamic programming.

Dynamic Programming

Step 1: The structure of an optimal parenthesization

- ➔ Let $A_{i..j}$, where $i \leq j$, denote the matrix that results from evaluating the product $A_i A_{i+1} \cdot \cdot \cdot A_j$. When $i < j$, we need to split the product into two products and compute $A_{i..k}$ and $A_{k+1..j}$ for some $i \leq k < j$, and then compute $A_{i..k}A_{k+1..j}$.
- ➔ The cost thus is the sum of the costs of compute $A_{i..k}$, $A_{k+1..j}$ and $A_{i..k}A_{k+1..j}$.

The optimal substructure of this problem is as follows.

Suppose that to optimally parenthesize $A_iA_{i+1} \cdot \cdot \cdot A_j$, we split the product between A_k and A_{k+1}. Then the subchain $A_iA_{i+1} \cdot \cdot \cdot A_k$ within this parenthesize must be optimal.

Otherwise, if we have a better parenthesize of $A_iA_{i+1} \cdot \cdot \cdot A_k$, then we can get a better parenthesize of $A_iA_{i+1} \cdot \cdot \cdot A_j$. By the similar argument, the parenthesize of $A_{k+1}A_{k+2} \cdot \cdot \cdot A_j$ is also optimal.

We can split the matrix-chain problem into two subproblems and find out the optimal solutions of these two subproblems.

Step 2: A recursive solution

- ➔ Let m[i, j] be the minimum number of scalar multiplications needed to compute $A_i A_{i+1} \cdots A_j$.
- ➔ We want to build m[i, j] recursively.

If we split $A_i A_{i+1} \cdots A_j$ between A_k and A_{k+1}, then the minimum costs will be $m[i, k] + m[k + 1, j] + p_{i-1} p_k p_j$

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

- The m[i, j] value gives the costs of optimal solution to problems, but they do not provide the construction of the optimal solution.
- We need to know the value of k which we used to split the product.
- We define s[i, j] to be a value of k at which we split the product $A_i A_{i+1} \cdots A_j$ in an optimal parenthesization.

Step 3: Computing the optimal costs

- ➔ From the recurrence, now the task is to find out the value m[1, n], which depends on m[i, j] for smaller chains with length $j - i$.
- ➔ So it is suitable to use the bottom-up method, i.e., start from computing m[i, j] for smaller $l = j - i$.
- ➔ Instead of use a recursive algorithm based on recurrence, we use a tabular, bottom-up method to compute the costs.

- We shall implement the tabular, bottom-up method in the procedure MATRIX-CHAIN-ORDER, which appears in the next slide.
- This procedure assumes that matrix A_i has dimensions $p_{i-1} \times p_i$ for i = 1,2,...,n. Its input is a sequence $p=<p_0$, p_1,...,p_n>, where p.length =n+1.
- The procedure uses an auxiliary table m[1..n, 1..n] for storing the m[i , j] costs and another auxiliary table s[1..n-1, 2..n] that records which index of k achieved the optimal cost in computing m[i, j].

```
MATRIX-CHAIN-ORDER(p)
1   $n = p.length - 1$
2   let $m[1..n, 1..n]$ and $s[1..n-1, 2..n]$ be new tables
3   for $i = 1$ to $n$
4       $m[i,i] = 0$
5   for $l = 2$ to $n$                    // $l$ is the chain length
6       for $i = 1$ to $n - l + 1$
7           $j = i + l - 1$
8           $m[i,j] = \infty$
9           for $k = i$ to $j - 1$
10              $q = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$
11              if $q < m[i,j]$
12                  $m[i,j] = q$
13                  $s[i,j] = k$
14  return $m$ and $s$
```

- The main cost for the procedure is the three nested for loops, which yields a running time of $O(n^3)$.
- We can prove that the running time is also $\Omega(n^3)$.
- The space requirement is $\Theta(n^2)$.

Step 4: Constructing an optimal solution

- ➔ Now we are able to give the optimal solution (the optimal parenthesizing the chain), because we have determined the value k in s[i, j].

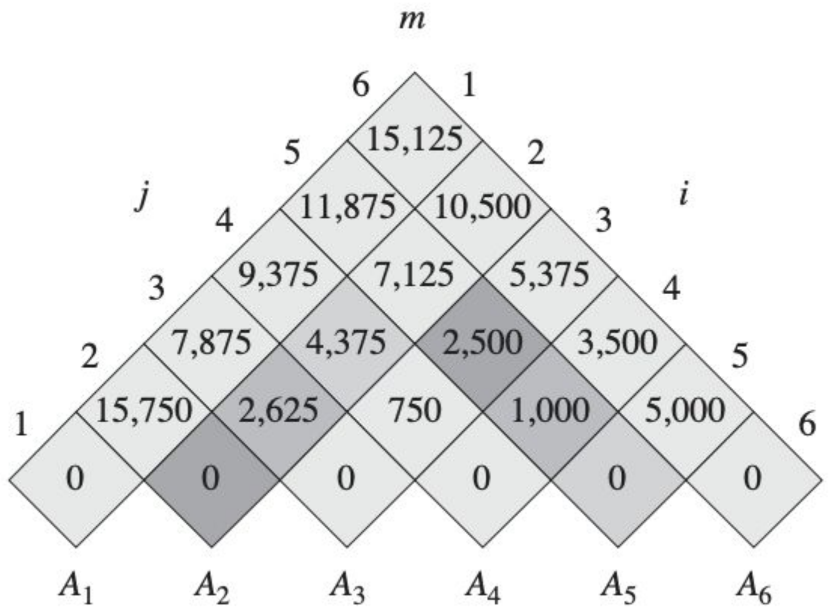
```
PRINT-OPTIMAL-PARENS($s, i, j$)
if $i == j$
    print “$A$”$_i$
else print “(”
    PRINT-OPTIMAL-PARENS($s, i, s[i, j]$)
    PRINT-OPTIMAL-PARENS($s, s[i, j] + 1, j$)
    print “)”
```

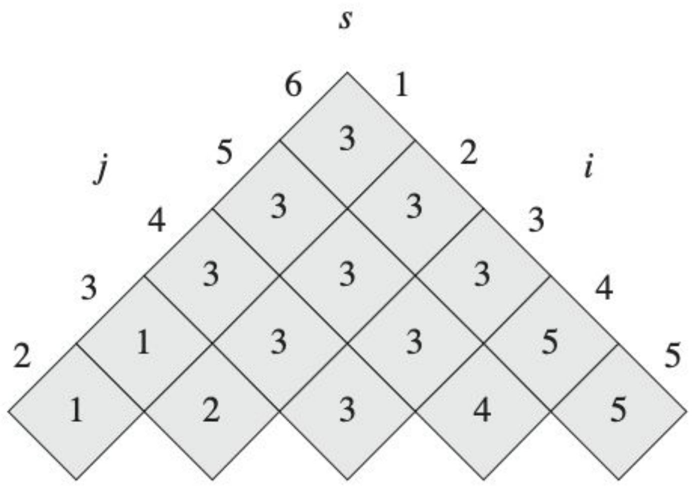
In the example, the call PRINT-OPTIMAL-PARENS(s, 1, 6) prints the parenthesization ((A1(A2A3))((A4A5)A6)).

An Example

Example: Suppose the following matrix-chain is given (p given)

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25





An Example

matrix	A_1	A_2	A_3	A_4	A_5	A_6
dimension	30×35	35×15	15×5	5×10	10×20	20×25

p0 p1 p2 p3 p4 p5 p6

$$m[2,5] = \min \begin{cases} m[2,2] + m[3,5] + p_1 p_2 p_5 = 0 + 2500 + 35 \cdot 15 \cdot 20 = 13{,}000\,, \\ m[2,3] + m[4,5] + p_1 p_3 p_5 = 2625 + 1000 + 35 \cdot 5 \cdot 20 = 7125\,, \\ m[2,4] + m[5,5] + p_1 p_4 p_5 = 4375 + 0 + 35 \cdot 10 \cdot 20 = 11{,}375 \end{cases}$$
$$= 7125\,.$$

Call Matrix-Chain-Order(p) and Print-Optimal-Parens(s, 1, 6) prints the parenthesization $((A_1(A_2A_3))((A_4A_5)A_6))$.

Example 4:

Longest Common Subsequence (LCS)

An Example

Biological applications often need to compare the DNA of two different organisms. A strand of DNA consists of a string of molecules called bases, where the possible bases are Adenine, Guanine, Cytosine and Thymine.

Usually DNA strands are expressed as a string over the finite set {A,C,G,T}. For example, the DNA of one organism may be

S1 = ACCGGTCGAGTGCGCGGAAGCCGGCCGAA,

Another organism may be

S2 = GTCGTTCGGAATGCCGTTGCTCTGTAAA.

One reason to compare two strands of DNA is to determine how closely related the two organisms are.

An Example

There are different ways to define the similarity of DNA strands. Here we consider one of the definitions.

The method is to find a subsequence S3 in which the bases in S3 appear in each of S1 and S2, these bases must appear in the same order, but not consecutively.

The longer the strand S3 we can find, the more similar S1 and S2 are. In the above example, the longest strand S3 is

GTCGTCGGAAGCCGGCCGAA.

Formal Definition of LCS

Formally, given a sequence X = $\langle x_1, x_2, \ldots, x_m \rangle$, and another sequence Z = $\langle z_1, z_2, \ldots, z_n \rangle$ is a subsequence of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of index of X such that $x_{i_j} = z_j$ for j = 1, 2, . . . , k.

For example, Z = ⟨B, C, D, B⟩ is a subsequence of X = ⟨A, B, C, B, D, A, B⟩ with corresponding index sequence ⟨2, 3, 5, 7⟩. Given two sequences X and Y , we say that a sequence Z is a common subsequence of X and Y if Z is a subsequence of both X and Y .

Formal Definition of LCS

Formally, given a sequence X[1..m], and another sequence Z[1..n] is a subsequence of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of index of X such that $X[i_j]=Z[j]$ for $j = 1, 2, \ldots, k$.

The LCS problem is to find a longest subsequence common to any two given sequences.

Brute-Force LCS Algorithm

For any two given sequences x[1..m] and y[1..n], check every subsequence of *x*[1 . . *m*] to see if it is also a subsequence of *y*[1 . . *n*].

Analysis

- 2^m subsequences of *x* (each bit-vector of length *m* determines a distinct subsequence of *x*).
- Hence, the runtime would be **exponential** !

Towards a Better Algorithm

Observation: Optimal Substructure of a LCS

Let x[1..m] and y[1..n] be sequences and z[1..k] be a LCS of x and y. Then,

1. If x[m]=y[n], then x[m]=y[n]=z[k] and z[1..k-1] is a LCS of x[1..m-1] and y[1..n-1].
2. If x[m]≠y[n], then x[m]≠z[k] implies that z is a LCS of x[1..m-1] and y[1..n].
3. If x[m]≠y[n], then y[n]≠z[k] implies that z is a LCS of x[1..m] and y[1..n-1].

DP Hallmark #1: Recursion

Observation: Optimal Substructure of a LCS

Let x[1..m] and y[1..n] be sequences and z[1..k] be a LCS of x and y. Then,

1. If x[m]=y[n], then x[m]=y[n]=z[k] and z[1..k-1] is a LCS of x[1..m-1] and y[1..n-1].
2. If x[m]≠y[n], then x[m]≠z[k] implies that z is a LCS of x[1..m-1] and y[1..n].
3. If x[m]≠y[n], then y[n]≠z[k] implies that z is a LCS of x[1..m] and y[1..n-1].

Towards a Better Algorithm

Using a similar idea for solving the matrix-chain product problem, we define c[i,j] be the length of an LCS of the sequence x[1..x] and y[1..j]. Then we have the following:

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

Dynamic Hallmark #1: Optimal Substructure

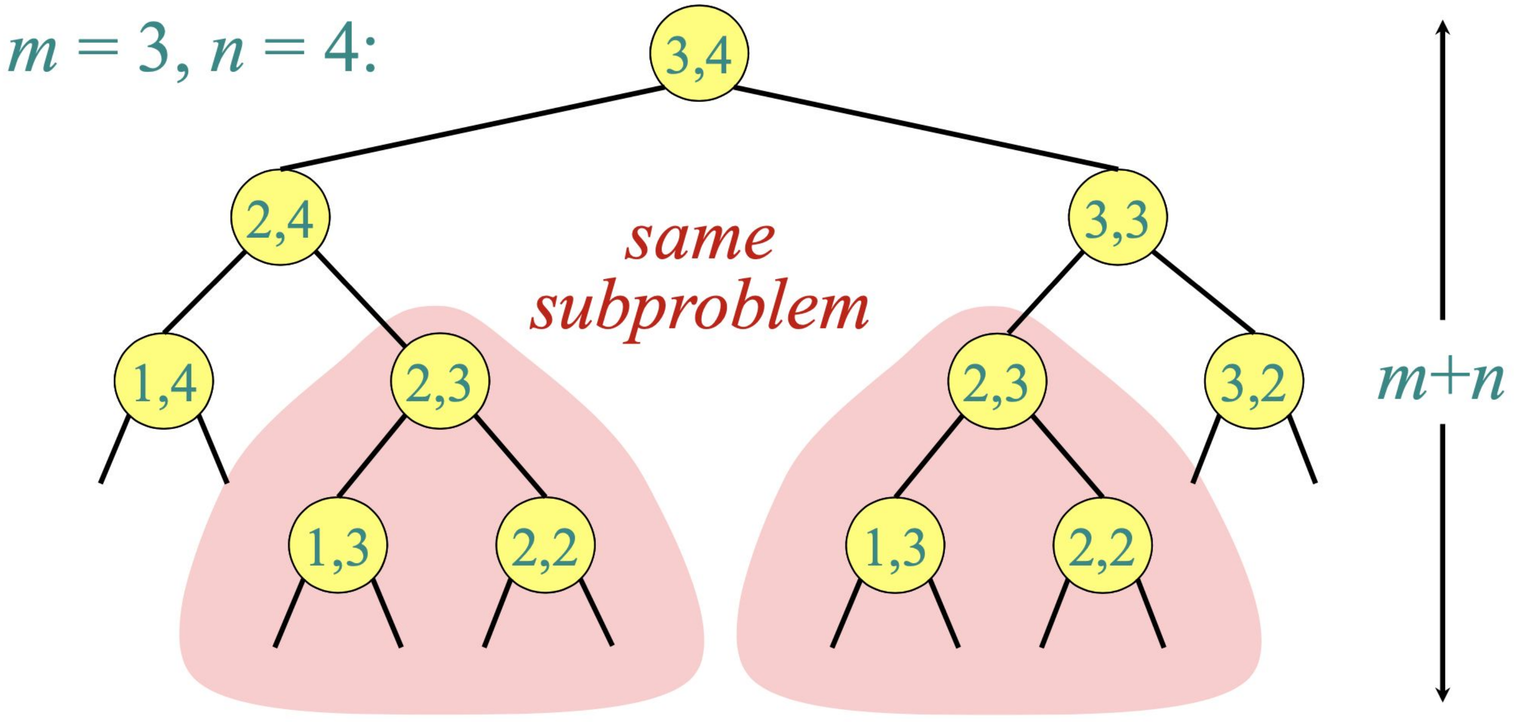
$\Rightarrow$ Recursion

Recursive Algorithm (not DP)

```
LCS(x,y, i,j)
    if (i=0 or j=0)
        c = 0
    else if (x[i]=y[j])
        c = LCS(x,y, i-1,j-1)+1
    else
        c = max{LCS(x,y, i,j-1), LCS(x,y, i-1,j)}
    return c
```

Worst-case: $x[i] \neq y[j]$, in which case the algorithm evaluates two subproblems, each with only one parameter decremented.

Recursion Tree



Run LCS(x,y,m,n)

Worst-case: *Height is m+n* and work potentially exponential, but we're solving subproblems already solved!

Dynamic-Programming Hallmark #2

Overlapping Subproblems

The distinct LCS subproblems are all the pairs (i,j). The number of such pairs for two strings of lengths m and n is only mn.

DP – Memoization algorithm

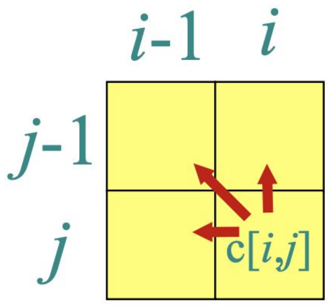
```
Memoized-LCS(x,y,m,n)
    Let c[m,n] be a new array
    For i=1 to m
        For j= 1 to n
            c[i,j] = - inf
    return Memoized-LCS-Aux(x,y,m,n)

Memoized-LCS-Aux(x,y,i,j)
    if (c[i,j] < 0)
        if (i=0 or j=0) c[i,j]=0
        else if x[i]=y[j]
            c[i,j]=LCS(x,y, i-1,j-1)+1
        else
            c[i,j]=max{LCS(x,y, i,j-1), LCS(x,y, i-1,j)}
    return c[i,j]
```

Space =
time =
$\Theta(mn)$

Bottom-Up DP Algorithm

$$c[i,j] = \begin{cases} \overset{\nwarrow}{c[i-1,j-1]} + 1 & \text{if } x[i] = y[j], \\ \max\{\underset{\leftarrow}{c[i-1,j]}, \underset{\uparrow}{c[i,j-1]}\} & \text{otherwise.} \end{cases}$$

c:

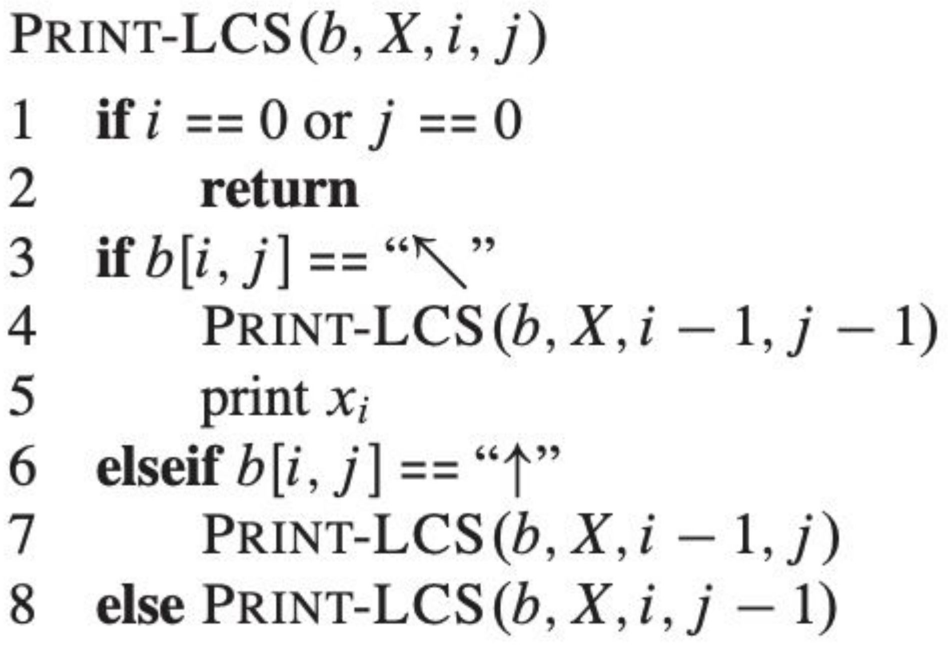


Bottom-Up DP Algorithm

```
LCS-LENGTH($X, Y$)
$m = X.length$
$n = Y.length$
let $b[1\,..\,m, 1\,..\,n]$ and $c[0\,..\,m, 0\,..\,n]$ be new tables
for $i = 1$ to $m$
    $c[i, 0] = 0$
for $j = 0$ to $n$
    $c[0, j] = 0$
for $i = 1$ to $m$
    for $j = 1$ to $n$
        if $x_i == y_j$
            $c[i, j] = c[i-1, j-1] + 1$
            $b[i, j] =$ "$\nwarrow$"
        elseif $c[i-1, j] \geq c[i, j-1]$
            $c[i, j] = c[i-1, j]$
            $b[i, j] = $ "$\uparrow$"
        else $c[i, j] = c[i, j-1]$
            $b[i, j] = $ "$\leftarrow$"
return $c$ and $b$
```

Space =
time =
$\Theta(mn)$

Bottom-Up DP Algorithm



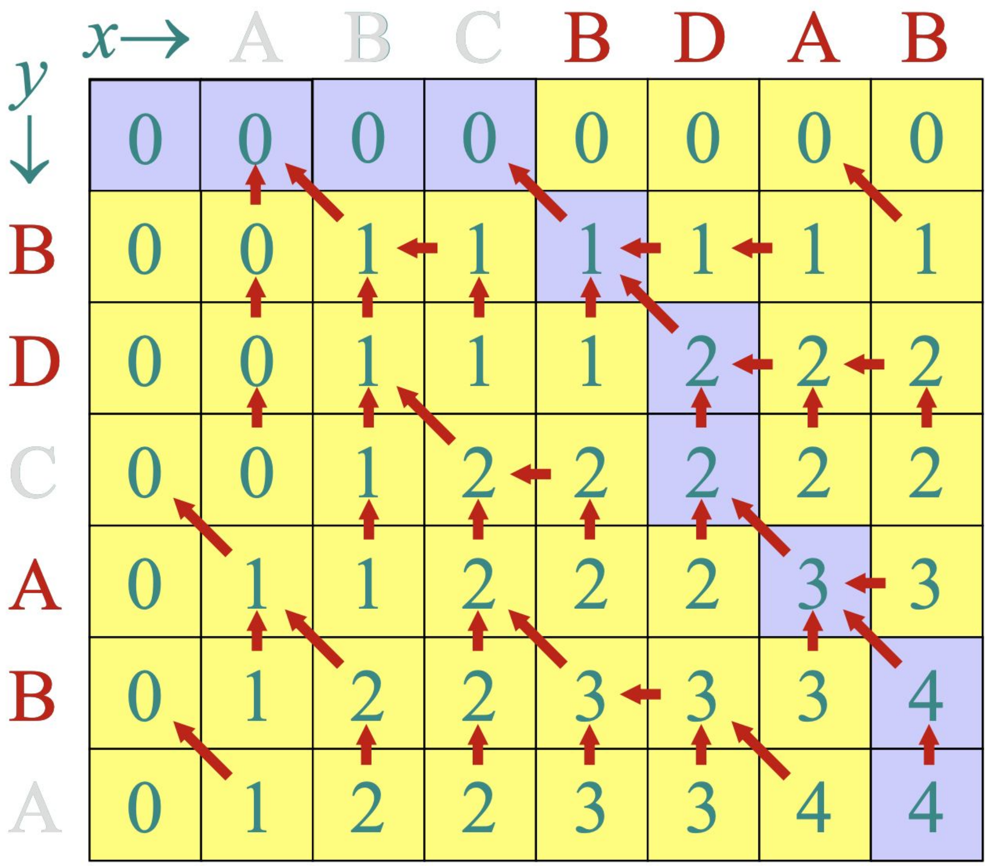
```
PRINT-LCS($b, X, i, j$)
if $i$ == 0 or $j$ == 0
        return
if $b[i, j]$ == "$\nwarrow$"
        PRINT-LCS($b, X, i-1, j-1$)
        print $x_i$
elseif $b[i, j]$ == "$\uparrow$"
        PRINT-LCS($b, X, i-1, j$)
else PRINT-LCS($b, X, i, j-1$)
```

time =
O(m+n)

This procedure prints LCS. The procedure takes time O(m+n), since it decrements at least one of i and j in each recursive call.

Bottom-Up DP Algorithm

- The procedure can be slightly improved, since just small part of the table b are useful for the solution. We can use just O(m + n) storage to store the useful part of the information of b.
- If we just need to know the length of LCS, then we can reduce the asymptotic space requirement.



$y \downarrow$	$x \rightarrow$	A	B	C	B	D	A	B
	0	0	0	0	0	0	0	0
B	0	0	1	1	1	1	1	1
D	0	0	1	1	1	2	2	2
C	0	0	1	2	2	2	2	2
A	0	1	1	2	2	2	3	3
B	0	1	2	2	3	3	3	4
A	0	1	2	2	3	3	4	4

Example 5:

Optimal Binary Search Trees

Optimal Binary Search Trees

Binary search tree is a binary tree, in which the keys in the left subtree is less than the key in the root while keys in the right subtree is greater than the key in the root, and a subtree of binary search tree is also a binary search tree.

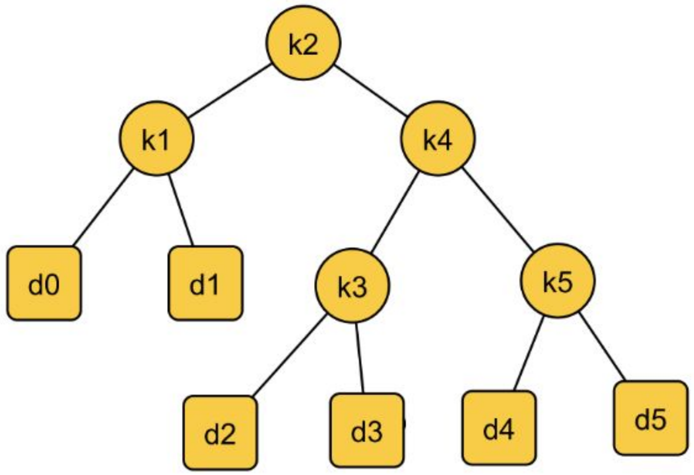
Now we consider a more general case. Suppose we have a sequence $K = \langle k_1, k_2, \dots, k_n \rangle$ of n distinct keys in sorted order (i.e., $k_1 < k_2 < \cdots < k_n$). For each key k_i, the probability a search will be on k_i is p_i. We wish to build a binary search tree for these keys such that the expected search time (the average search time) is optimal.

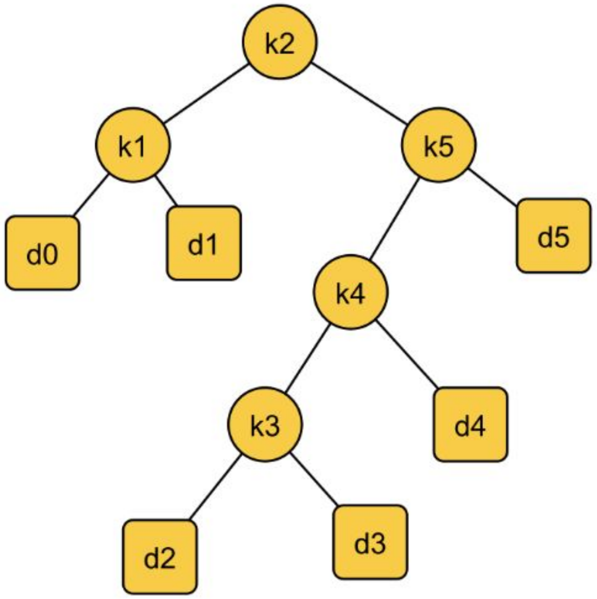
Optimal Binary Search Trees

We also need to consider the search values that are not in K. So we have n+1 dummy keys d_0, d_1, d_2 . . . d_n, where, d_i, 0<i<n, represents the values between k_i and k_{i+1}, d_0 represents the values less than k_1 and d_n represents the values greater than k_n. For each dummy key d_j, we assume the probability for searching according to it is q_j. For each dummy key d_j, we assume the probability for searching according to it is q_j. So, we have

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

Optimal Binary Search Trees





Optimal Binary Search Trees

Suppose we have already established the binary search tree T (in the tree, dummy keys should be leaves). Then we have the expected cost of a search in T is

$$\begin{aligned} E[\text{search cost in } T] &= \sum_{i=1}^{n} (depth_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (depth_T(d_i) + 1) \cdot q_i \\ &= 1 + \sum_{i=1}^{n} depth_T(k_i) \cdot p_i + \sum_{i=0}^{n} depth_T(d_i) \cdot q_i , \end{aligned}$$

where *depthT* denotes a node's depth in the tree T. If the expected search cost is the smallest, then we call T an optimal binary search tree.

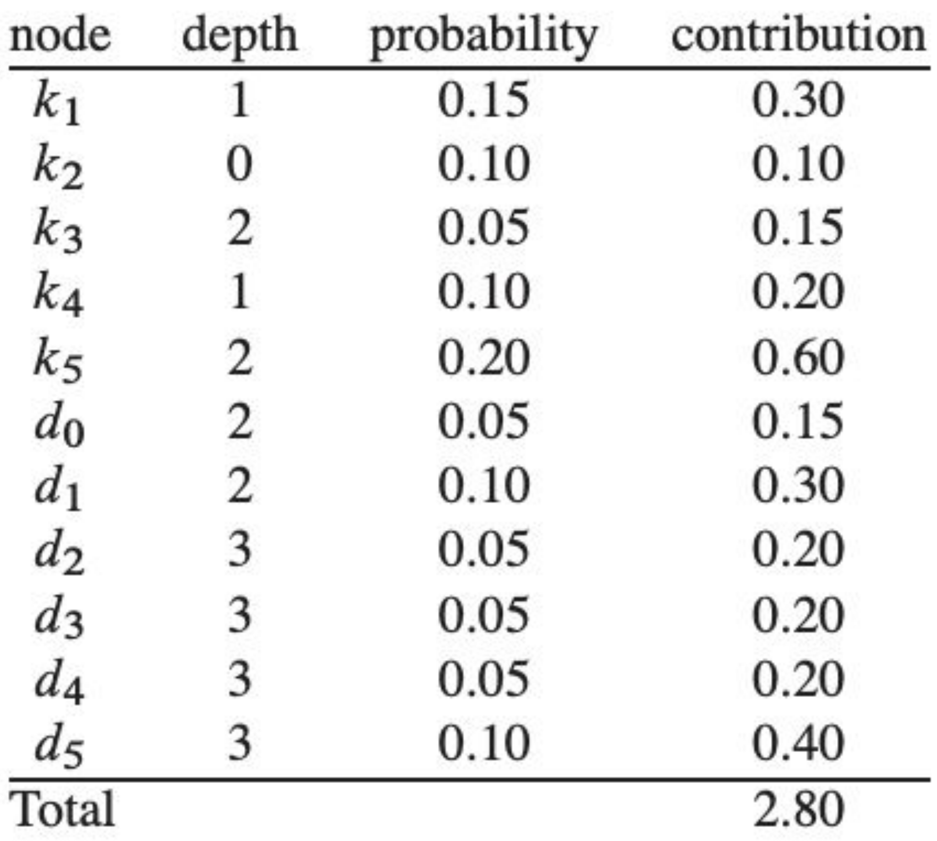
Optimal Binary Search Trees

i	0	1	2	3	4	5
p_i		0.15	0.10	0.05	0.10	0.20
q_i	0.05	0.10	0.05	0.05	0.05	0.10

Two binary search trees are displayed in the previous slide.

The first tree has the expected search cost 2.80 and the second tree has the expected search cost 2.75, which is optimal.

Optimal Binary Search Trees



node	depth	probability	contribution
k_1	1	0.15	0.30
k_2	0	0.10	0.10
k_3	2	0.05	0.15
k_4	1	0.10	0.20
k_5	2	0.20	0.60
d_0	2	0.05	0.15
d_1	2	0.10	0.30
d_2	3	0.05	0.20
d_3	3	0.05	0.20
d_4	3	0.05	0.20
d_5	3	0.10	0.40
Total			2.80

Optimal Binary Search Trees

To construct the tree, we can first construct a binary search tree with the n keys, then add the dummy nodes to leaves. But the number of binary search tree with n nodes is $\Theta(4^n/n^{3/2})$. So exhaustive search is not feasible. We can consider to use dynamic programming.

Dynamic Programming

Step 1: The structure of an optimal binary search tree

Suppose we have constructed an optimal binary search tree. Then each subtree must contain keys in a contiguous range $k_i, k_{i+1}, \ldots, k_j$, for some $1 \le i \le j \le n$. In addition, that subtree must also contains the leaves of dummy keys $d_{i-1}, d_i, \ldots, d_j$.

Therefore we have the optimal substructure: if an optimal binary search tree T has a subtree T′ containing keys $k_i, \ldots, k_j$, then T′ must be optimal as well for subproblem with keys $k_i, \ldots, k_j$ and dummy keys $d_{i-1}, \ldots, d_j$. Otherwise we can replace the subtree with better expected cost and that means that T is not optimal.

Dynamic Programming

Considering the recursive method, if a subtree contains keys $k_i, \ldots, k_j$ and the root is k_r, then the left subtree contains keys $k_i, \ldots, k_{r-1}$ (and dummy keys $d_{i-1}, \ldots, d_{r-1}$) and the right subtree contains keys $k_{r+1}, \ldots, k_j$ (and dummy keys $d_r, \ldots, d_j$). When the root is i, then the left subtree contains only d_{i-1} and when k_j is the root, its right subtree contains only d_j . We may try every possible key as the root to obtain the optimal subtree.

Dynamic Programming

Step 2: A recursive solution

We can define the values of optimal solution for subtrees as follows. For a subtree with keys $k_i , . . . , k_j$, define $e[i, j]$ to be the optimal expected cost of searching, where $i \geq 1, i - 1 \leq j \leq n$. Here we define $e[i, i - 1]$ as the subtree with d_{i-1} as a only node. So

$$e[i, i - 1] = q_{i-1}.$$

Dynamic Programming

- When j ≥ i, we need to select a root k_r, which forms two subtrees, one with the keys $d_i, \ldots, d_{r-1}$ and another with the keys $d_{r+1}, \ldots, d_j$.
- For a tree containing keys $k_s, \ldots, k_t$, the optimal value is e[s, t].
- But when it becomes a subtree, the depth of each vertex will increase one. Therefore the the expected costs for this subtree will be

$$e[s,t] + \sum_{l=s}^{t} p_l + \sum_{l=s-1}^{t} q_l.$$

Dynamic Programming

We define

$$w(s,t) = \sum_{l=s}^{t} p_l + \sum_{l=s-1}^{t} q_l$$

Thus, if k_r is the root of an optimal subtree containing keys k_i ,..., k_j , we have

$$e[i,j] = p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j)) .$$

Note that $w(i,j) = w(i,r-1) + p_r + w(r+1,j)$

We have $e[i,j] = e[i,r-1] + e[r+1,j] + w(i,j) .$

Dynamic Programming

Now we have the recursive formula for e[i, j].

$$e[i,j] = \begin{cases} q_{i-1} & \text{if } j = i-1 \\ \min_{i \le r \le j}\{e[i,r-1] + e[r+1,j] + w(i,j)\} & \text{if } i \le j. \end{cases}$$

To help us to keep the track of the structure of optimal binary search tree, we define root[i, j] to be the index r for which k_r is the root of an optimal binary search tree containing keys k_i ,. . . , k_j .

Dynamic Programming

Step 3: Computing the expected search cost of an optimal BST

Similar to other dynamic programming, we need to use some tables to store the solutions for subproblems. So we define tables *e*, *w* and *root* in the following procedure. For *e* and *w* we need to define $1 \le i \le n + 1$, $0 \le j \le n$, because we need to record the values of “empty” subtrees (e.g., $e[i, i - 1], 1 \le i \le n$).

Dynamic Programming

```
OPTIMAL-BST($p, q, n$)
let $e[1..n+1, 0..n]$, $w[1..n+1, 0..n]$,
        and $root[1..n, 1..n]$ be new tables
for $i = 1$ to $n + 1$
    $e[i, i-1] = q_{i-1}$
    $w[i, i-1] = q_{i-1}$
for $l = 1$ to $n$
    for $i = 1$ to $n - l + 1$
        $j = i + l - 1$
        $e[i, j] = \infty$
        $w[i, j] = w[i, j-1] + p_j + q_j$
        for $r = i$ to $j$
            $t = e[i, r-1] + e[r+1, j] + w[i, j]$
            if $t < e[i, j]$
                $e[i, j] = t$
                $root[i, j] = r$
return $e$ and $root$
```

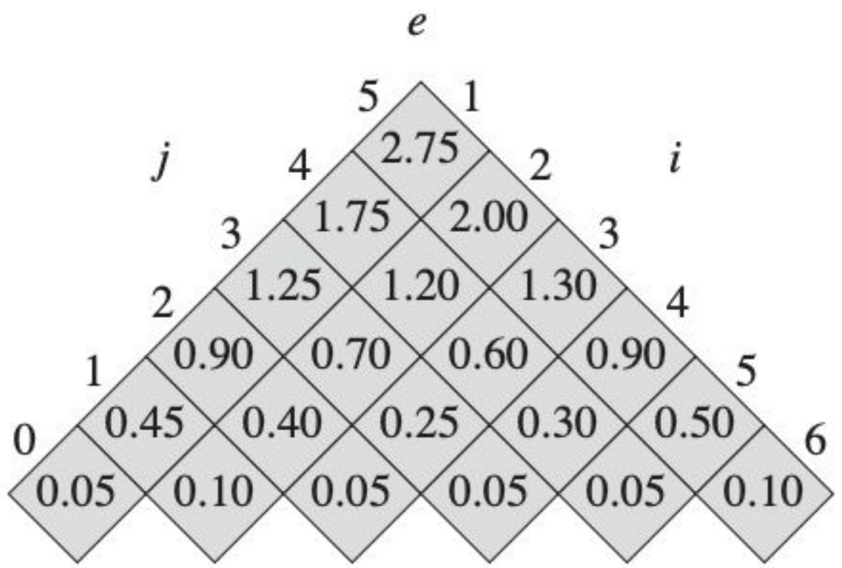
Running Time Discussion

The Optimal-BST procedure takes $\Theta(n^3)$ time.

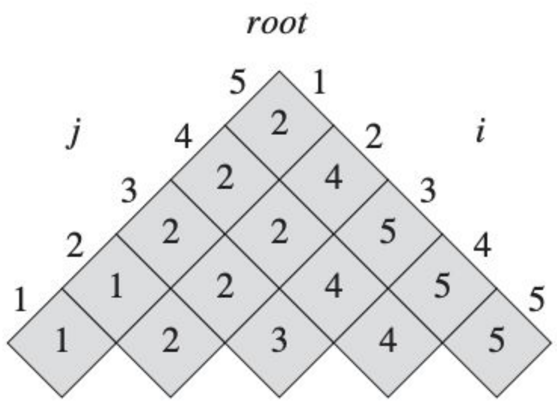
Because the main costs are the three nested for loops, each loop index takes at most n values, the running time is $O(n^3)$.

On the other hand, we can also see that the procedure takes $\Omega(n^3)$ time.

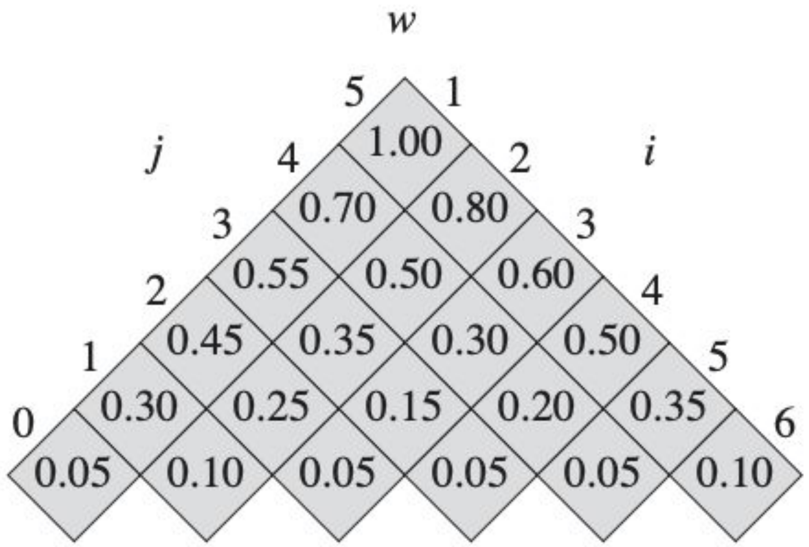
Example



e
j
i
0.05 0.10 0.05 0.05 0.05 0.10
0.45 0.40 0.25 0.30 0.50
0.90 0.70 0.60 0.90
1.25 1.20 1.30
1.75 2.00
2.75



root
j
i
1 2 3 4 5
1 2 4 5
2 2 5
2 4
2



w
j
i
0.05 0.10 0.05 0.05 0.05 0.10
0.30 0.25 0.15 0.20 0.35
0.45 0.35 0.30 0.50
0.55 0.50 0.60
0.70 0.80
1.00

After Class

- After class:

  read Part IV Chapter 15.4-15.5