Introduction to Algorithms

Topic 6-1 : Dynamic Programming

Xiang-Yang Li and Haisheng Tan

School of Computer Science and Technology
University of Science and Technology of China (USTC)

Fall Semester 2025

Outline

Rod Cutting

Matrix-chain Multiplication

Elements of Dynamic Programming

Longest Common Subsequence

Optimal Binary Search Trees

Dynamic Programming

Dynamic programming, like the divide-and-conquer method, solves problems by combining the solutions to subproblems.

We typically apply dynamic programming to optimization problems. Such problems can have many possible solutions. Each solution has a value, and we wish to find a solution with the optimal (minimum or maximum) value. We call such a solution an optimal solution to the problem, as opposed to the optimal solution, since there may be several solutions that achieve the optimal value.

Dynamic Programming

When developing a dynamic-programming algorithm, we follow a sequence of four steps:

- Characterize the structure of an optimal solution.
- Recursively define the value of an optimal solution.
- Compute the value of an optimal solution, typically in a bottom-up fashion.
- Construct an optimal solution from computed information. (optional)

Contents

Rod Cutting
- Problem Description
- Recursive Top-down Implementation
- Memoization
- Bottom-up Version
- Reconstructing a Solution

Matrix-chain Multiplication

Elements of Dynamic Programming

Longest Common Subsequence

Optimal Binary Search Trees

Rod Cutting

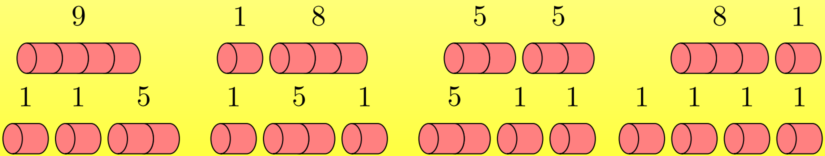
Problem Description

Given a rod of length n inches and a table of prices p_i for $i = 1, 2, \ldots, n$, determine the maximum revenue r_n obtainable by cutting up the rod and selling the pieces. Note that if the price p_n for a rod of length n is large enough, an optimal solution may require no cutting at all.

- p_i is the price of rod of length i.
- A feasible solution: $n = i_1 + i_2 + \cdots + i_m$, where i_j is a positive integer.
- Revenue $r_n = \sum_{j=1}^{m} p_{i_j}$.

Example of Rod Cutting Problem

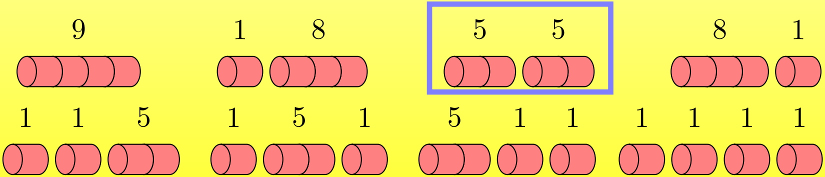
length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30



All the cases for $n = 4$.

Example of Rod Cutting Problem

length i	1	2	3	4	5	6	7	8	9	10
price p_i	1	5	8	9	10	17	17	20	24	30



All the cases for $n = 4$.

Analysis

We view a decomposition as consisting of a first piece of length i cut off the left-hand end, and then a right-hand remainder of length $n-i$. Only the remainder, and not the first piece, may be further divided. We may view every decomposition of a length-n rod in this way: as a first piece followed by some decomposition of the remainder. We thus obtain the following equation:

$$r_n = \max_{1 \le i \le n} (p_i + r_{n-i}).$$

Recursive Top-down Implementation

```
Cut-Rod(p, n)
if n == 0 then return 0
q = −∞
for i = 1 to n do
    q = max(q, p[i] + Cut-Rod(p, n − i))
return q
```

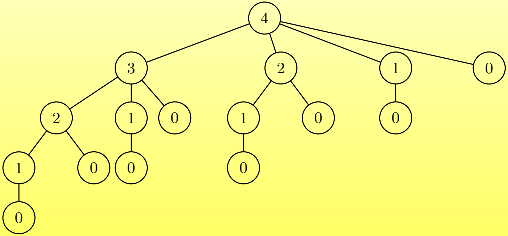
Recursive Top-down Implementation

```
Cut-Rod(p, n)
if n == 0 then return 0
q = −∞
for i = 1 to n do
    q = max(q, p[i] + Cut-Rod(p, n − i))
return q
```

Cut-Rod is Inefficient

The problem is that Cut-Rod calls itself recursively over and over again with the same parameter values, i.e., it solves the same subproblems repeatedly.

Cut-Rod is Inefficient



Example: n=4.

2^n nodes, 2^{n-1} leaves.

Let T(n) denote the total number of calls made to Cut-Rod when called with its second parameter equal to n. We have $T(0) = 1$ and $T(n) = 1 + \sum_{j=0}^{n-1} T(j)$. That is

$$T(n) = 2^n.$$

Top-down with Memoization

```
Memoized-Cut-Rod(p, n)
let r[0..n] be a new array
for i = 0 to n do r[i] = −∞
return Memoized-Cut-Rod-Aux(p, n, r)

Memoized-Cut-Rod-Aux(p, n, r)
if r[n] ≥ 0 then return r[n]     // check whether r[n] has been calculated.
if n == 0 then
    q = 0
else
    q = −∞
    for i = 1 to n do
        q = max(q, p[i] + Memoized-Cut-Rod-Aux(p, n − i, r))
r[n] = q
return q
```

Bottom-up Version

```
Bottom-Up-Cut-Rod(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n do
    q = −∞
    for i = 1 to j do
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

Bottom-up Version

```
Bottom-Up-Cut-Rod(p, n)
let r[0..n] be a new array
r[0] = 0
for j = 1 to n do
    q = −∞
    for i = 1 to j do
        q = max(q, p[i] + r[j − i])
    r[j] = q
return r[n]
```

The bottom-up and top-down versions have the same asymptotic running time $\Theta(n^2)$.

Reconstructing a Solution

```
Extended-Bottom-Up-Cut-Rod(p, n)
   // Record the optimal value computed for each subproblem,
   and a choice that led to the optimal value
   let r[0..n] and s[0..n] be new arrays
   r[0] = 0
   for j = 1 to n do
       q = −∞
       for i = 1 to j do
           if q < p[i] + r[j − i] then
               q = p[i] + r[j − i]
               s[j] = i
       r[j] = q
   return r and s
```

Reconstructing a Solution

Print-Cut-Rod-Solution(p, n)

1: (r, s) = Extended-Bottom-Up-Cut-Rod(p, n)
2: while $n > 0$ do
3:     print s[n]
4:     $n = n - s[n]$

Contents

Rod Cutting

Matrix-chain Multiplication
- Problem Description
- Solution

Elements of Dynamic Programming

Longest Common Subsequence

Optimal Binary Search Trees

Problem Description

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 $A_1 A_2 ... A_n$ in a way that minimizes the number of scalar multiplications.

Note that in the matrix-chain multiplication problem, we are not actually multiplying matrices. Our goal is only to determine an order for multiplying matrices that has the lowest cost.

Step 1: The Structure of an Optimal Parenthesization

For convenience, let us adopt the notation $A_{i..j}$, where $i \leq j$, for the matrix that results from evaluating the product $A_i A_{i+1} \ldots A_j$.

When $i < j$, any parenthesization of the product $A_i A_{i+1} \ldots A_j$ must split the product between A_k and A_{k+1} for some integer k in the range $i \leq k < j$.

Step 1: The Structure of an Optimal Parenthesization

The optimal substructure of the optimal parenthesization problem is as follows:

If an optimal parenthesization of $A_iA_{i+1}...A_j$ splits the product between A_k and A_{k+1}, the parenthesization of the "prefix" subchain $A_iA_{i+1}...A_k$ within this optimal parenthesization of $A_iA_{i+1}...A_j$ must be an optimal parenthesization of $A_iA_{i+1}...A_k$.

Thus, we can build an optimal solution to an instance of the matrix-chain multiplication problem by splitting the problem into two subproblems.

Step 2: A Recursive Solution

The subproblems are to determining the minimum cost of a parenthesization of $A_i A_{i+1} \ldots A_j$ for $1 \leq i \leq j \leq n$.

Let m[i,j] be the minimum number of scalar multiplications needed to compute the matrix $A_{i..j}$, so $m[1,n]$ is the cost of the solution for the full problem.

Obtain the recursive equation of m[i,j] by the following analysis:

- If $i = j$, the chain consists of just one matrix $A_{i..i} = A_i, m[i,i] = 0$.
- If $i < j$, assumed that the optimal parenthesization splits the product $A_i A_{i+1} \ldots A_j$ between A_k and $A_{k+1}, i \leq k < j$, and each matrix is $p_{i-1} \times p_i$, thus $m[i,j] = m[i,k] + m[k+1,j] + p_{i-1} p_k p_j$.

Step 2: A Recursive Solution

So we obtain:

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

For the full problem, $m[1,n]$ is the cost of the optimal solution.

In order to keep track of how to construct an optimal solution, we define $s[i,j]$ to be a value of k at which we can split the product $A_iA_{i+1}...A_j$ to obtain an optimal parenthesization $s[i,j] = k$, such that:

$$m[i,j] = m[i,k] + m[k+1,j] + p_{i-1}p_kp_j$$

Step 3: Computing the Optimal Costs

There are relatively few subproblems: one problem for each choice of i and j satisfying $1 \le i \le j \le n$, or $\binom{n}{2} + n = \Theta\left(n^2\right)$ in all.

But each subproblems may be encountered many times in different branches of the recursion tree.

We use a tabular, bottom-up approach to compute the optimal cost.

Step 3: Computing the Optimal Costs

The following pseudocode assumes that matrix A_i has dimensions $p_{i-1} \times p_i$ for $i = 1, 2, ..., n$.

The input is a sequence $p = \langle p_0, p_1, \ldots, p_n \rangle$, where $length[p] = n + 1$.

The procedure uses an auxiliary table $m[1..n, 1..n]$ for storing the $m[i,j]$ costs.

An auxiliary table $s[1..n-1, 2..n]$ records which index of k achieved the optimal cost in computing $m[i,j]$ and it will be used to construct an optimal solution.

Because the cost $m[i,j]$ depends only on the costs of computing matrix-chain products of fewer than $j - i + 1$ matrices, the table m will be filled in a manner that corresponds to solving the parenthesization problem on matrix chains of increasing length.

Step 3: Computing the Optimal Costs

MATRIX-CHAIN-ORDER(p)

```
n = p.length − 1
let m[1..n, 1..n] and s[1..n − 1, 2..n] be new tables
for i = 1 to n do
    m[i, i] = 0
for l = 2 to n do
    for i = 1 to n − l + 1 do
        j = i + l − 1, m[i, j] = ∞
        for k = i to j − 1 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
return m and s
```

Step 3: Computing the Optimal Costs

Figure 1 illustrates this procedure on a chain of $n = 6$ matrices.

Since the definition of m[i,j] is only for $i \leq j$, only the portion of the table m strictly above the main diagonal is used.

The figure shows the table rotated to make the main diagonal run horizontally.

The matrix chain is listed along the bottom.

The minimum cost m[i,j] can be found at the intersection of lines running northeast from A_j and northwest from A_i.

Each horizontal row in the table contains the entries for matrix chains of the same length.

MATRIX-CHAIN-ORDER computes the rows from bottom to top and from left to right within each row.

An entry m[i,j] is computed using the products $p_{i-1}p_kp_j$ for $k = i, i+1, ..., j-1$ and all entries southwest and southeast from m[i,j].

Step 3: Computing the Optimal Costs

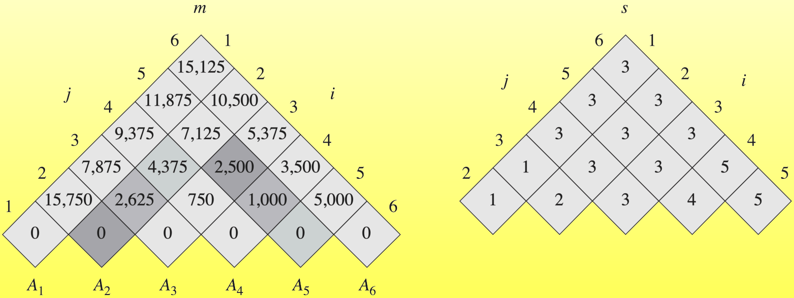


Figure: The m and s tables computed by MATRIX-CHAIN-ORDER for n = 6 and the following matrix dimensions:

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

Step 3: Computing the Optimal Costs

Computing $m[2,5]$:

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

$$= 7125$$

The minimum number of scalar multiplications to multiply the 6 matrices is $m[1,6] = 15,125$.

The running time of MATRIX-CHAIN-ORDER is $\Omega(n^3)$ and it requires $\Theta(n^2)$ space to store the m and s tables.

Thus, MATRIX-CHAIN-ORDER is much more efficient than the exponential-time method.

Step 4: Constructing an Optimal Solution

An optimal solution can be constructed from the computed information stored in the table $s[1...n, 1...n]$

Each entry $s[i,j]$ records the value of k such that the optimal parenthesization of $A_i A_{i+1} ... A_j$ splits the product between A_k and A_{k+1}.

Thus the final matrix multiplication in computing $A_{1..n}$ optimally is $A_{1..s[1,n]}$ $A_{s[1,n]+1..n}$ and the earlier matrix multiplications can be computed recursively based on $s[1, n]$, since $s[1, s[1, n]]$ determines the last matrix multiplication in computing $A_{1..s[1,n]}$, and $s[s[1, n] + 1, n]$ determines the last matrix multiplication in computing $A_{s[1,n]+1..n}$.

Step 4: Constructing an Optimal Solution

The following procedure prints an optimal parenthesization of $\langle A_i, A_{i+1}, \ldots, A_j \rangle$, given the s table computed by MATRIX-CHAIN-ORDER and the indices i and j.

```
PRINT-OPTIMAL-PARENS(s,i,j)
if i = j then
    print "A_i"
else
    print "("
    PRINT-OPTIMAL-PARENS(s,i,s[i,j])
    PRINT-OPTIMAL-PARENS(s,s[i,j]+1,j)
    print ")"
```

The initial call PRINT-OPTIMAL-PARENS$(s, 1, n)$ prints an optimal parenthesization of $\langle A_i, A_{i+1}, \ldots, A_j \rangle$.

In Figure 1, the call PRINT-OPTIMAL-PARENS$(s, 1, 6)$ prints the parenthesization $((A_1 (A_2 A_3)) ((A_4 A_5) A_6))$.

Contents

Rod Cutting

Matrix-chain Multiplication

Elements of Dynamic Programming
- Optimal Substructure
- Overlapping Subproblems

Longest Common Subsequence

Optimal Binary Search Trees

Two Key Ingredients

An optimization problem must have two key ingredients so that it can apply dynamic programming:

optimal substructure and overlapping subproblems.
Time-memory Trade-off.

Optimal Substructure

- A problem exhibits optimal substructure: optimal solutions to a problem incorporate optimal solutions to related subproblems, which we may solve independently.
- Whenever a problem exhibits optimal substructure, we have a good clue that dynamic programming might apply.
- In dynamic programming, we build an optimal solution to the problem from optimal solutions to subproblems.

Discovering Optimal Substructure

1. A solution to the problem consists of making a choice, such as choosing an initial cut in a rod (Rod Cutting) or choosing an index at which to split the matrix chain (Matrix-chain Multiplication). Making this choice leaves one or more subproblems to be solved.
2. Supposing that for a given problem, you are given the choice that leads to an optimal solution. You do not concern yourself yet with how to determine this choice. You just assume that it has been given to you.
3. Given this choice, you determine which subproblems ensue and how to best characterize the resulting space of subproblems.
4. You show that the solutions to the subproblems used within an optimal solution to the problem must themselves be optimal by using a "cut-and-paste" technique.

Overlapping Subproblems

- Typically, the total number of distinct subproblems is a polynomial in the input size. When a recursive algorithm revisits the same problem repeatedly, we say that the optimization problem has overlapping subproblems.
- In contrast, a problem for which a divide-and-conquer approach is suitable usually generates brand-new problems at each step of the recursion.
- Dynamic-programming algorithms typically take advantage of overlapping subproblems by solving each subproblem once and then storing the solution in a table where it can be looked up when needed, using constant time per lookup.

Example: Recursive-Matrix-Chain

Recursive-Matrix-Chain(p, i, j)

if $i == j$ then
    return 0
$m[i,j] = \infty$
for $k = i$ to $j-1$ do
    $q =$ Recursive-Matrix-Chain(p, i, k)
        +Recursive-Matrix-Chain(p, k+1, j)
        $+p_{i-1}p_k p_j$
    if $q < m[i,j]$ then
        $m[i,j] = q$
return $m[i,j]$

Recursion Tree of Recursive-Matrix-Chain(p, 1, 4)

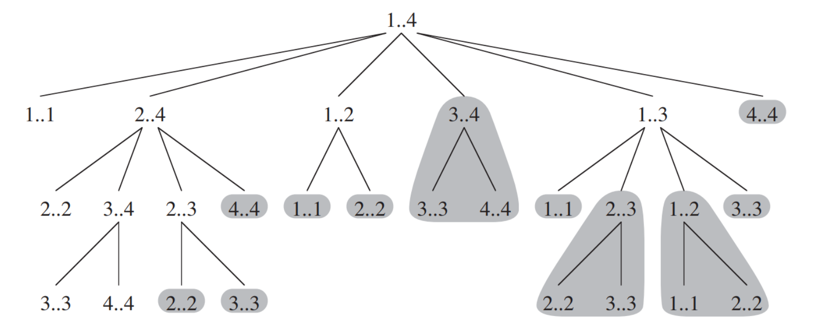


Figure: The recursion tree for the computation of Recursive-Matrix-Chain(p, 1, 4)

Memoized-Matrix-Chain

```
Memoized-Matrix-Chain(p)
n = p.length − 1
let m[1..n, 1..n] be a new table.
for i = 1 to n do
    for j = i to n do
        m[i,j] = ∞
return Lookup-Chain(m, p, 1, n)

Lookup-Chain(m, p, i, j)
if m[i,j] < ∞ then
    return m[i,j]
if i == j then
    m[i,j] = 0
else
    for k = i to j − 1 do
        q = Recursive-Matrix-Chain(p, i, k)
          +Recursive-Matrix-Chain(p, k + 1, j) + p_{i−1} p_k p_j
        if q < m[i,j] then
            m[i,j] = q
return m[i,j]
```

Contents

Rod Cutting

Matrix-chain Multiplication

Elements of Dynamic Programming

Longest Common Subsequence
- Problem Description
- Solution

Optimal Binary Search Trees

Problem Description

Subsequence

Given a sequence $X = \langle x_1, x_2, ..., x_m \rangle$, another sequence $Z = \langle z_1, z_2, ..., z_k \rangle$ is a subsequence of X if there exists a strictly increasing sequence $\langle i_1, i_2, ..., i_k \rangle$ of indices of X such that for all $j = 1, 2, ..., k$, we have $x_{i_j} = z_j$

Example: $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$ with corresponding index sequence $\langle 2, 3, 5, 7 \rangle$

Problem Description

Subsequence

Given a sequence $X = \langle x_1, x_2, \ldots, x_m \rangle$, another sequence $Z = \langle z_1, z_2, \ldots, z_k \rangle$ is a subsequence of X if there exists a strictly increasing sequence $\langle i_1, i_2, \ldots, i_k \rangle$ of indices of X such that for all $j = 1, 2, \ldots, k$, we have $x_{i_j} = z_j$

Example: $Z = \langle B, C, D, B \rangle$ is a subsequence of $X = \langle A, B, C, B, D, A, B \rangle$ with corresponding index sequence $\langle 2, 3, 5, 7 \rangle$

Common Subsequence

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.

Problem Description

Problem Description:longest-common-subsequence(LCS)

Given two sequences $X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$,we wish to find a maximum-length common subsequence of X and Y.

Step 1: Characterizing a longest common subsequence

the i th prefix of X

Given a sequence $X = \langle x_1, x_2, ..., x_m \rangle$, we define the i th prefix of X, for $i = 0, 1, ..., m$, as $X_i = \langle x_1, x_2, ..., x_i \rangle$.

Step 1: Characterizing a longest common subsequence

the i th prefix of X

Given a sequence $X = \langle x_1, x_2, ..., x_m \rangle$, we define the i th prefix of X, for $i = 0, 1, ..., m$, as $X_i = \langle x_1, x_2, ..., x_i \rangle$.

Theorem 15.1 (Optimal substructure of an LCS)

Let $X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$ be sequences, and let $Z = \langle z_1, z_2, ..., z_k \rangle$ be any LCS of X and Y.

- if $x_m = y_n$, then $z_k = x_m = y_n$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.
- if $x_m \neq y_n$, then $z_k \neq x_m$ implies that Z is an LCS of X_{m-1} and Y.
- if $x_m \neq y_n$, then $z_k \neq y_n$ implies that Z is an LCS of X and Y_{n-1}.

Step 1: Characterizing a longest common subsequence

Proof

1) If $z_k \neq x_m$, then we could append $x_m = y_n$ to Z to obtain a common subsequence of X and Y of length $k+1$, contradicting the supposition. Thus, $z_k = x_m = y_n$ Suppose a common subsequence W of X_{m-1} and Y_{n-1} with length greater than $k-1$.Then, appending $x_m = y_n$ to W can produce a contradiction.
2) If $z_k \neq x_m$, then Z is a commom subsequence of X_{m-1} and Y. If there were a common subsequence W of X_{m-1} and Y with length greater than k,then W would also be a commmon subsequence X_m and Y, contradicting the assumption.
3) The proof is symmetric to 2).

Step 2: A recursive solution

- if $x_m = y_n$, we must find an LCS of X_{m-1} and Y_{n-1},then appending $x_m = y_n$ to this LCS yields an LCS of X and Y.
- if $x_m \neq y_n$, two subproblems must be solved: finding an LCS of X_{m-1} and Y and finding an LCS of X and Y_{n-1}.The longer one is the answer.
- Let $c[i,j]$ denote the length of an LCS of the sequence X_i and Y_j

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

Step 3: Computing the length of an LCS

- Let b[i,j] points to the table entry corresponding to the optimal subproblem solution chosen when computing c[i,j].

$$b[i,j] = \begin{cases} \nwarrow & \text{if } c[i,j] \text{ is decided by } c[i-1,j-1] \\ \uparrow & \text{if } c[i,j] \text{ is decided by } c[i-1,j] \\ \leftarrow & \text{if } c[i,j] \text{ is decided by } c[i,j-1] \end{cases}$$

- A dynamic programming algorithm,LCS-LENGTH,computes the length of an LCS of two sequences,$X = \langle x_1, x_2, ..., x_m \rangle$ and $Y = \langle y_1, y_2, ..., y_n \rangle$.
- The procedure returns the b and c tables; c[m,n] contains the length of an LCS of X and Y.

Step 3: Computing the length of an LCS

```
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 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
            b[i, j] = "↖"
        else if c[i − 1, j] ≥ c[i, j − 1] then
            c[i, j] = c[i − 1, j]
            b[i, j] = "↑"
        else
            c[i, j] = c[i, j − 1]
            b[i, j] = "←"
return c and b
```

Step 3: Computing the length of an LCS

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	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						
7	B	0						

Step 3: Computing the length of an LCS

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0						
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Step 3: Computing the length of an LCS

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0						
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Step 3: Computing the length of an LCS

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0						
5	D	0						
6	A	0						
7	B	0						

Step 3: Computing the length of an LCS

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Step 3: Computing the length of an LCS

- The tables produced by LCS-LENGTH. Inputs are: $X = \langle A, B, C, B, D, A, B \rangle$ $Y = \langle B, D, C, A, B, A \rangle$.
- Since each table entry takes $O(1)$ time to compute, the running time of the procedure is $O(mn)$.

Step 4: Constructing an LCS

- The b table returned by LCS-LENGTH can be used to construct an LCS.
- We begin at b[m, n] and trace through the table following the arrows.
- A "$\nwarrow$" in entry b[i,j] implies that $x_i = y_j$ is an element of the LCS.
- The elements of the LCS are encountered in reverse order by this method.

Step 4: Constructing an LCS

	j	0	1	2	3	4	5	6
i		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Step 4: Constructing an LCS

i	j	0	1	2	3	4	5	6
		y_j	B	D	C	A	B	A
0	x_i	0	0	0	0	0	0	0
1	A	0	↑ 0	↑ 0	↑ 0	↖ 1	← 1	↖ 1
2	B	0	↖ 1	← 1	← 1	↑ 1	↖ 2	← 2
3	C	0	↑ 1	↑ 1	↖ 2	← 2	↑ 2	↑ 2
4	B	0	↖ 1	↑ 1	↑ 2	↑ 2	↖ 3	← 3
5	D	0	↑ 1	↖ 2	↑ 2	↑ 2	↑ 3	↑ 3
6	A	0	↑ 1	↑ 2	↑ 2	↖ 3	↑ 3	↖ 4
7	B	0	↖ 1	↑ 2	↑ 2	↑ 3	↖ 4	↑ 4

Step 4: Constructing an LCS

```
PRINT-LCS(b,X,i,j)
if i == 0 or j == 0 then
    return
if b[i,j] == "↖" then
    PRINT-LCS(b,X,i−1,j−1)
    print x_i
else if b[i,j] == "↑" then
    PRINT-LCS(b,X,i−1,j)
else
    PRINT-LCS(b,X,i,j−1)
```

- The left recursive procedure prints out an LCS of X and Y in the proper, forward order
- The initial call is PRINT-LCS$(b, X, X.length, Y.length)$.
- The procedure takes time $O(m+n)$, since at least one of i and j is decremented in each stage of the recursion.

Step 4: Constructing an LCS

Improving the code

- We can improve dynamic-programming algorithms on the time or space it uses.
- Some changes can simplify the code and improve constant factors but otherwise yield no asymptotic improvement in performance.
- Others can yield substantial asymptotic savings in time and space.

Contents

Rod Cutting

Matrix-chain Multiplication

Elements of Dynamic Programming

Longest Common Subsequence

Optimal Binary Search Trees
- Problem Description
- Solution

Optimal Binary Search Trees

Problem Description

Formally, we are given a sequence $K = \langle k_1, k_2, \ldots, k_n \rangle$ of n distinct keys in sorted order (so that $k_1 < k_2 < \cdots < k_n$), and we wish to build a binary search tree from these keys. For each key k_i, we have a probability p_i that a search will be for k_i.

Some searches may be for values not in K, and so we also have $n+1$ "dummy keys" $d_0, d_1, \ldots, d_n$ representing values not in K. In particular, d_0 represents all values less than k_1, d_n represents all values greater than k_n, and for $i = 1, 2, \ldots, n-1$, the dummy key d_i represents all values between k_i and k_{i+1}. For each dummy key d_i, we have a probability q_i that a search will correspond to d_i.

Optimal Binary Search Trees

Problem Description

Each key k_i is an internal node, and each dummy key d_i is a leaf. Every search is either successful (finding some key k_i) or unsuccessful (finding some dummy key d_i), and so we have $\sum_{i=1}^{n} p_i + \sum_{i=0}^{n} q_i = 1$. The cost of a search is set as the number of nodes examined. The expected cost of a search in T is

$$E[\text{search cost in } T] = \sum_{i=1}^{n} (\text{depth}_T(k_i) + 1) \cdot p_i + \sum_{i=0}^{n} (\text{depth}_T(d_i) + 1) \cdot q_i$$

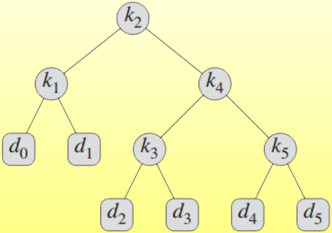
Optimal Binary Search Trees

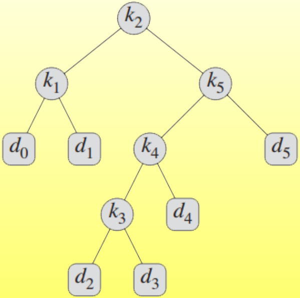
Problem Description

The expected cost of a search in T is

$$\mathrm{E}[\text{search cost in T}] = 1 + \sum_{i=1}^{n} \mathrm{depth}_T(k_i) \cdot p_i + \sum_{i=0}^{n} \mathrm{depth}_T(d_i) \cdot q_i.$$

For a given set of probabilities, we wish to construct a binary search tree whose expected search cost is the smallest. We call such a tree an optimal binary search tree.





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

The expected search cost of the left one is 2.80.
The expected search cost of the right one is 2.75. This tree is optimal.

Step 1: The structure of an optimal binary search tree

- Any subtree of a binary search tree must contain keys in a contiguous range $k_i, \ldots, k_j$, for some $1 \le i \le j \le n$. A subtree that contains keys $k_i, \ldots, k_j$ must also have as its leaves the dummy keys $d_{i-1}, \ldots, d_j$.
- The optimal substructure is: if an optimal binary search tree T has a subtree T' containing keys $k_i, \ldots, k_j$ then this subtree T' must be optimal as well for the subproblem with keys $k_i, \ldots, k_j$ and dummy keys $d_{i-1}, \ldots, d_j$. Its correctness can be proved by "cut-and-paste" argument.
- We can construct an optimal solution to the problem from optimal solution to subproblems.

Step 1: The structure of an optimal binary search tree

- Given keys $k_i, \ldots, k_j$ one of these keys, say $k_r (i \leq r \leq j)$, will be the root of an optimal subtree containing these keys. The left subtree of the root k_r contains the keys $k_i, \ldots, k_{r-1}$ and the right subtree contains the keys $k_{r+1}, \ldots, k_j$.
- Examining all candidate roots k_r and determining all optimal binary search trees containing and those containing $k_{r+1}, \ldots, k_j$. Then we will find an optimal binary search tree.
- One detail worth noting about "empty" subtrees: for keys $k_i, \ldots, k_j$, we select k_i as the root. Then a subtree containing keys k_i has no actual keys but does contain the single dummy key d_{i-1}. The case of choosing k_j as the root is symmetrical.

Step 2: A recursive solution

- Our subproblem is to finding an optimal binary search tree containing the keys $k_i, \ldots, k_j$, where $i \geq 1, j \leq n$, and $j \geq i-1$.
- Define $e[i,j]$ as the expected cost of searching an optimal binary search tree containing the keys. Our ultimate goal is $e[1,n]$.
- We discuss different cases to obtain the recurrence of $e[i,j]$
  - When $j = i-1$, we have just the dummy key d_{i-1}, the expected search cost is $e[i,i-1] = q_{i-1}$.
  - When $j \geq i$, select a root k_r, $i \leq r \leq j$.
  - The expected search cost of this subtree increases by the sum of all the probabilities in the subtree.

Step 2: A recursive solution

- This sum of probabilities for a subtree with key $k_i, \ldots, k_j$ is $w(i,j) = \sum_{l=i}^{j} p_l + \sum_{l=i-1}^{j} q_l$.
- $e[i,j] = p_r + (e[i,r-1] + w(i,r-1)) + (e[r+1,j] + w(r+1,j))$
- Note that $w(i,j) = w(i,r-1) + p_r + w(r+1,j)$, so we have

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

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

Step 2: A recursive solution

- The e[i,j] values give the expected search costs in optimal binary search trees.
- To keep track of the structure of optimal binary search trees, we define root[i,j], for $1 \leq i \leq j \leq n$, to be the index r for which is the root k_r of an optimal binary search tree containing keys.

Step 3: Computing the expected search cost

- We store the $e[i,j]$ values in a table $e[1..n+1, 0..n]$.
- The first index needs to run to $n+1$ because existing a subtree containing only the dummy key d_n and we need to compute and store $e[n+1,n]$. The second index needs to start from 0 because in order to have a subtree containing only the dummy key d_0 and compute and store $e[1,0]$.
- We use a table $root[i,j]$, for recording the root of the subtree containing keys $k_i, \ldots, k_j$.
- We store $w(i,j)$ in a table $w[1..n+1, 0..n]$, where

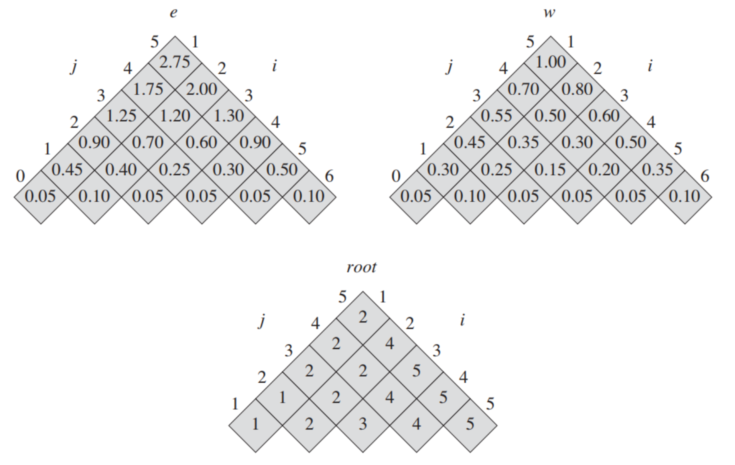
$$w[i,j] = w[i,j-1] + p_j + q_j.$$

- Thus, we can compute the $\Theta(n^2)$ values of $w[i,j]$ in $\Theta(1)$ time each.

Optimal-BST

```
Optimal-BST(p, q, n)
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 and root
```

Tables computed by Optimal-BST



Time complexity of Optimal-BST

- In the Optimal-BST procedure, for loops are nested three loops and each loop index takes on at most n values.
- The loop indices do not have the same bounds as those in Matrix-Chain-Order, but they are within at most 1 in all directions.
- Thus, the Optimal-BST procedure takes $\Theta(n^3)$ time, like Matrix-Chain-Order.