Efficient Algorithms

Ekkapot Charoenwanit

Software Systems Engineering
TGGS
KMUTNB

Lecture 7: Dynamic Programming

Fibonacci Numbers

Fibonacci numbers can be defined as a recurrence as follows:

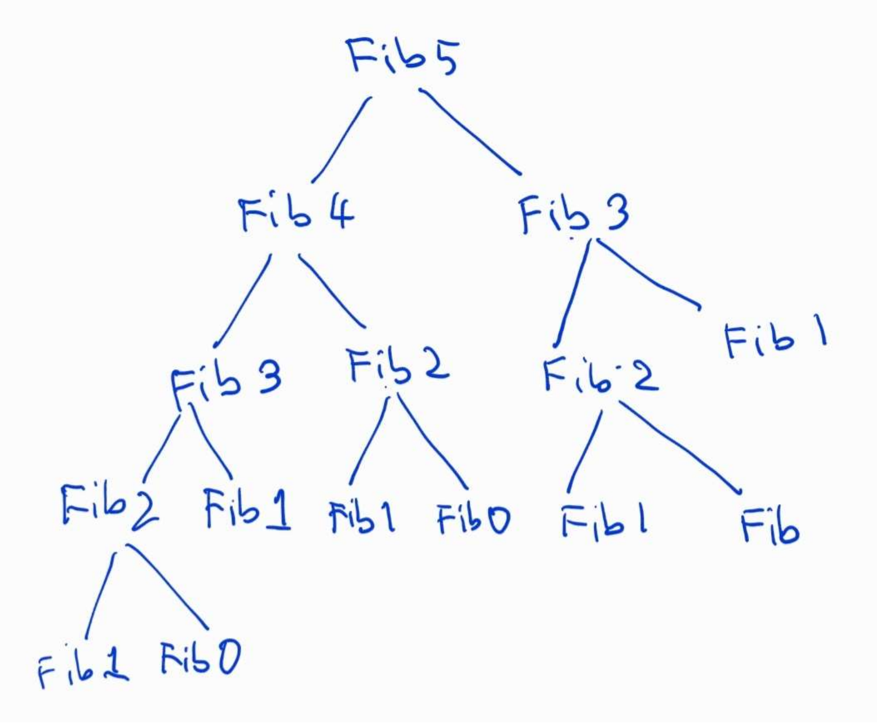
$$F_0 = 0$$
$$F_1 = 1$$
$$F_n = F_{n-1} + F_{n-2} \text{ for } n \geq 2$$

Fibonacci Numbers: Top-Down Approach

Top-Down Approach

- implemented via recursion

```
procedure FIB(n)
    if n ≤ 1 then
        return n
    else
        FIB(n − 1) + FIB(n − 2)
```



Fibonacci Numbers: Top-Down Approach

Let $N(n)$ be the number of recursive calls $FIB(n)$ makes.

We can write $N(n)$ as

$$N(n) = N(n-1) + N(n-2) + 1 \text{ for } n \geq 2$$

$$N(0) = 1 \text{ and } N(1) = 1$$

Solving the recurrence, we have

$$N(n) = 2F(n+1) - 1$$

,where $F(n) = \frac{\phi^n - (1-\phi)^n}{\sqrt{5}}$,where $\phi = \frac{1+\sqrt{5}}{2} \approx 1.61803$

Fibonacci Numbers: Top-Down Approach

We can express the running time as

$$T(n) = T(n-1) + T(n-2) + c$$

Because $T(n)$ is a non-decreasing function,

$$T(n) \geq 2T(n-2) + c$$

Therefore, we have

$$T(n-2) \geq 2T(n-4) + c$$

$$T(n) \geq 2\{2T(n-4) + c\} + c$$

$$= 2^2T(n-4) + 2^1c + 2^0c$$

$$T(n) \geq 2^2\{2T(n-6) + c\} + 2^1c + 2^0c$$

$$= 2^3T(n-6) + 2^2c + 2^1c + 2^0c$$

Keep Expanding until the k^{th} term:

$$T(n) \geq 2^{k-1}\{2T(n-2k) + c\} + 2^{k-2}c + \cdots + 2^0c$$

$$= 2^kT(n-2k) + 2^{k-1}c + 2^{k-2}c + \cdots + 2^0c$$

Fibonacci Numbers: Top-Down Approach

Recursion terminates when $n - 2k = 0$.

Therefore, $n = 2k$ or $k = \frac{n}{2}$.

$$\begin{aligned} T(n) \geq\ & 2^k T(0) + 2^{k-1}c + \cdots + 2^0 c \\ =\ & 2^k c + 2^{k-1}c + \cdots + 2^0 c \\ =\ & c\,\frac{1(2^{k+1}-1)}{2-1} = c\left(2 \cdot 2^{n/2} - 1\right) = c(2 \cdot \sqrt{2}^n - 1) \end{aligned}$$

Therefore,

$$T(n) \in \Omega(\sqrt{2}^n)$$

This proves that the running time of the top-down approach is ***at least*** exponential in the value of n. ■

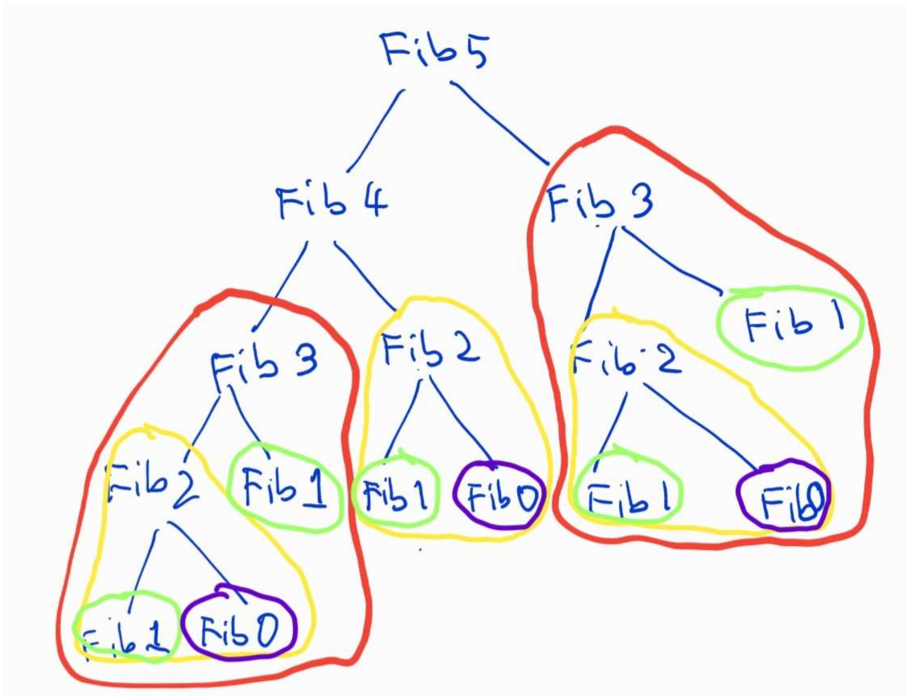
Space complexity is proportional to the depth of the recursion tree, i.e., $\Theta(n)$. ■

Fibonacci Numbers: Overlapping Subproblems

The recursion tree of $FIB(n)$ exhibits a property known as ***overlapping subproblems***.

As you can see in the recursion tree for $FIB(5)$,

- $FIB(0)$ is called 3 times
- $FIB(1)$ is called 5 times
- $FIB(2)$ is called 3 times
- $FIB(3)$ is called 2 times
- $FIB(4)$ is called 1 time



Fibonacci Numbers: Memoization

Memoization means "***to remember***" by storing values into a ***table***.

All the entries of the table *F* are initially initialized to *0*.

- The number of different subproblems is $n+1$.
- The number of recursive calls (non-memoized) is $\Theta(n)$.

```
procedure FIB(n,F[1...n])
    if n ≤ 1 then
        return n
    else
        if F[n] > 0 then
            return F[n]
        F[n] = FIB(n − 1, F) + FIB(n − 2, F)
        return F[n]
```

Fibonacci Numbers: Memoization

Therefore, $T(n-2) = \Theta(1)$ with ***memoization***.

The total running time is composed of the time $T(n-1)$ for ***recursive call*** $FIB(n-1)$, the time $T(n-2)$ for ***memorized call*** $FIB(n-2)$ and the time $\Theta(1)$ for ***non-recursive work*** in each call.

$$T(n) = T(n-1) + \Theta(1) + \Theta(1)$$

, which can be simplified as

$$T(n) = T(n-1) + \Theta(1)$$

Solving using the repeated substitution method, we have

$$T(n) = \Theta(n)$$

Fibonacci Numbers: Bottom-Up Approach

<u>Bottom-Up Approach</u>:

- Implemented via tabulation
- Smaller values -> Larger values
- Space complexity is $\Theta(n)$
- Time complexity is $\Theta(n)$

```
procedure FIB(n)
    F = NEW TABLE[0...n]
    F[0] = 0
    F[1] = 1
    for i = 2 → n do
        F[i] = F[i − 1] + F[i − 2]
    return F[n]
```

***Space complexity can be further optimized. (See PS 5.1.2)

Top-Down vs Bottom-Up

***Top-Down Approach*:**

- Divides the original problem into smaller subproblems
- Solves the subproblems
- Combines the solutions to subproblems
- Is implemented via recursion
- Minimizes the number of subproblems via memoization

***Bottom-Up Approach*:**

- Solves smaller subproblems first
- Generates solutions to larger subproblems from solutions to smaller ones
- Is implemented using loop iterations
- Solves each subproblem only once

Dynamic Programming

Dynamic programming (DP) is an algorithm ***design paradigm*** where a problem is solved recursively by dividing into smaller subproblems like divide-and-conquer.

However, DP exploits the following ***two properties*** of the problem:

- ***Overlapping subproblems***
  - Subproblems solving the same problems are repeated many times
  - Solve such repeated subproblems only once and reuse their solutions via
    - Memoization (top-down)
    - Tabulation (bottom-up)
- ***Optimal Substructure***
  - An optimal solution to a problem contains optimal solutions to smaller subproblems

DP is usually used to solve ***optimization problems*** as we will see shortly.

- ***Minimization***
- ***Maximization***

Matrix Chain Multiplication

Recalling matrix multiplication:

1) The product $C = AB$ of a $p \times q$ matrix A and a $q \times r$ matrix B is a $p \times r$ matrix C computed by

$$c_{ij} = \sum_{k=1}^{q} a_{ik} \cdot b_{kj}$$

for $1 \leq i \leq p$ and $1 \leq j \leq r$.

2) Matrix multiplication is ***associative***, i.e., $A(BC) = (AB)C$ so different ***parenthesizations*** do not alter the value.

Matrix Chain Multiplication

$$C \quad = \quad A \quad \times \quad B$$

$$[p \times r] \quad [p \times q] \quad [q \times r]$$

To multiply A and B,

pqr multiplications are needed.

Matrix Chain Multiplication: Example

Example: Let A_1, A_2 and A_3 be matrices of the following dimensions 10×100, 100×5 and 5×50, respectively.

There are ***2*** different parenthesizations for a matrix chain of length ***3***.

Case I: $A_1(A_2\ A_3)$
A_2A_3 requires $100 \cdot 5 \cdot 50 = 25000$ multiplications, whose result is a matrix $A_{2,3}$ of dimension 100×50.
$A_1A_{2,3}$ requires $10 \cdot 100 \cdot 50 = 50000$ multiplications, whose result is a matrix $A_{1,3}$ of dimension 10×50.
Therefore, the total number of multiplications is $25000 + 50000 = 75000$.

Case II: $(A_1A_2)\ A_3$
A_1A_2 requires $10 \cdot 100 \cdot 5 = 5000$ multiplications, whose result is a matrix $A_{1,2}$ of dimension 10×5.
$A_{1,2}A_3$ requires $10 \cdot 5 \cdot 50 = 2500$ multiplications, whose result is a matrix $A_{1,3}$ of dimension 10×50.
Therefore, the total number of multiplications is $5000 + 2500 = 7500$.

Matrix Chain Multiplication: Brute Force

How many different ways can we parenthesize a matrix chain of length n?

Let $C(n)$ denote the number of ways of parenthesizations of a matrix chain of length n.

Therefore, $C(n) = \begin{cases} 1, & n \leq 2 \\ \sum_{k=1}^{n-1} C(k)\, C(n-k), & n \geq 3 \end{cases}$

$C(n) = \Omega(\frac{4^n}{n^{1.5}})$, which is related to the Catalan numbers.

Since the number of ways of placing parenthesis is ***exponential*** in the length of a matrix chain, a brute force approach is impractical.

Matrix Chain Multiplication: Notation

Given a matrix chain $A_1 A_2 \dots A_n$ of length n,

A_1 has a dimension of $p_0 \times p_1$,

A_2 has a dimension of $p_1 \times p_2$,

…

A_n has a dimension of $p_{n-1} \times p_n$.

$A_i \dots A_j$ is a matrix of size $p_{i-1} \times p_j$.

Let $m(i,j)$ be the number of multiplications for $A_i \dots A_j$.

We want to find $m(1,n)$.

Matrix Chain Multiplication: Optimal Substructure

Suppose we ***split*** a matrix chain $A_i \ldots A_j$ at some position $\mathrm{i} \leq k < j$.

$$A_i \ldots A_j = (A_i \ldots A_k)(A_{k+1} \ldots A_j)$$

$A_i \ldots A_k$ evaluates to a $p_{i-1} \times p_k$ matrix $A_{i,k}$ whose number of multiplications is $m(i,k)$.

$A_{k+1} \ldots A_j$ evaluates to a $p_k \times p_j$ matrix $A_{(k+1),j}$ whose number of multiplications is $m(k+1,j)$.

Multiplying A_{ik} and $A_{(k+1),j}$ requires $p_{i-1}p_k p_j$ multiplications.

Therefore, the total number of multiplications is $m(i,k) + m(k+1,j) + p_{i-1}p_k p_j$.

Matrix Chain Multiplication: Optimal Substructure

If an optimal solution to $A_i \dots A_j$ involves splitting into $A_i \dots\ A_k$ and $A_{k+1} \dots A_j$ at the final step, solutions to parenthesizations of $A_i \dots\ A_k$ and $A_{k+1} \dots A_j$ are also optimal.

Proof: We will use a ***Cut-and-Paste*** argument.

Suppose the solution $m(i, k)$ to $A_i \dots\ A_k$ is ***not optimal***. We can then replace this solution to $A_i \dots\ A_k$ with a ***better solution*** $m'(i, k) < m(i, k)$, hence a ***better solution*** to $A_i \dots A_j$

$$m'(i, j) = m'\ (i, k) + m(k + 1, j) + p_{i-1}p_k p_j < m(i, j)$$

,which contradicts the optimality of the solution to $A_i \dots A_j$.

A symmetric argument is applied to optimality of $A_{k+1} \dots A_j$. ■

Matrix Chain Multiplication: Optimal Substructure

Having proved ***optimal substructure*** of the Matrix Chain Multiplication problem,

the next question is "***where*** do we split ?, i.e., what is the position k ?".

Answer: Try them all !!!

Matrix Chain Multiplication: Recursive Formulation

Recursive Formulation: Let $M(i,j)$ be the minimum number of multiplications for a matrix chain $A_i \dots A_j$.

$$M(i,j) = \begin{cases} 1, & i = j \\ \min\limits_{i \le k < j} \{M(i,k) + M(k+1,j)\} + p_{i-1}p_k p_j & i < j \end{cases}$$

We do not know k so we try all the $j - i$ possible values and find k that provides the ***best*** (***smallest***) solution.

Matrix Chain Multiplication: Top-Down

```
procedure MCM(i,j,p[0...n])
    if i == j then
        return 0
    minMCM = ∞
    for k = i → j − 1 do
        minMCM = min(minMCM,
                     MCM(i, k, p) + MCM(k + 1, j, p) + p[i] ∗ p[k] ∗ p[j])
    return minMCM
```

The running time is exponential in the chain length n.

Matrix Chain Multiplication: Overlapping Subproblems

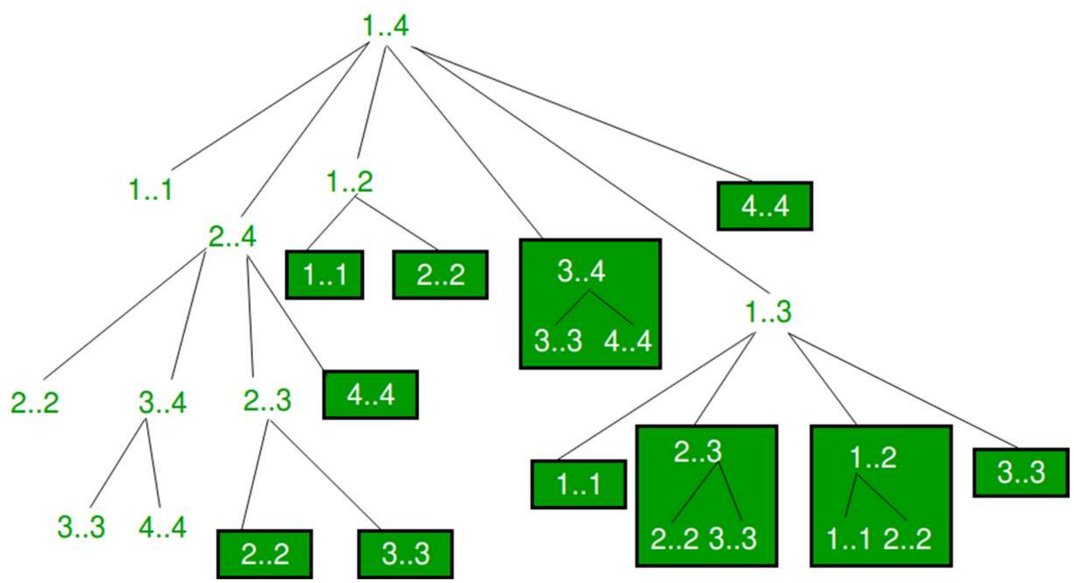


***Illustration from https://www.geeksforgeeks.org/

Matrix Chain Multiplication: Overlapping Subproblems

Additionally, the number of ***distinct*** subproblems is relatively ***small*** (i.e. ***polynomial*** in problem size).

The number of distinct subproblems is

$\Theta(n^2)$, which is polynomial in the problem size n.

Matrix Chain Multiplication: Memoization

```
1:  procedure MCM(i,j,p[0...n], M[1...n][1...n])
2:      if i == j then
3:          return 0
4:      if M[i][j] > 0 then
5:          return M[i][j]
6:      M[i][j] = ∞
7:      for k = i → j − 1 do
8:          M[i][j] = min(M[i][j],
9:                  MCM(i, k, p, M) + MCM(k + 1, j, p, M) + p[i] * p[k] * p[j])
10:     return M[i][j]
```

This ***top-down memoized*** algorithm remains similar to the ***top-down non-memoized*** algorithm, except that this memorized algorithm stores newly computed values into the table M as shown in lines 8 and 9 and reuses these values as shown in lines 4 and 5.

Matrix Chain Multiplication: Bottom-Up

<u>Analysis (Rough Version)</u>:

There are $\Theta(n^2)$ distinct subproblems (generated by the two outer for loops).

In each subproblem, there are $\Theta(n)$ ways of choosing where to split the matrix chain (the innermost for loop in line 10).

Therefore, the total running time is $\Theta(n^3)$.

```
1:  procedure MCM(p[0...n])
2:      M = NEW TABLE[0...n][0...n]
3:      P = NEW TABLE[1...n − 1][2...n]
4:      for i = 1 → n do
5:          M[i][i] = 0
6:      for l = 2 → n do
7:          for i = 1 → n − l + 1 do
8:              j = i − l + 1
9:              M[i][j] = ∞
10:             for k = i → j − 1 do
11:                 q = M[i, k] + M[k + 1, j] + p[i] ∗ p[k] ∗ p[j]
12:                 if q < M[i][j] then
13:                     M[i][j] = q
14:                     S[i][j] = k
15:     return (M, S)
```

Matrix Chain Multiplication: Solution Reconstruction

```
procedure PrintOptimalParen(i, j, S)
    if i == j then
        print A_i
    else
        print (
        PrintOptimalParen(i, S[i][j], S)
        PrintOptimalParen(S[i][j] + 1, j, S)
        print )
```

Longest Common Subsequence

Definition: The ***Longest Common Subsequence*** (***LCS***) problem is as follows. Given two strings X of length m and Y of length n, our goal is to determine the longest common subsequence, that is, the longest sequence of characters that do not necessarily appear contiguously in the two strings.

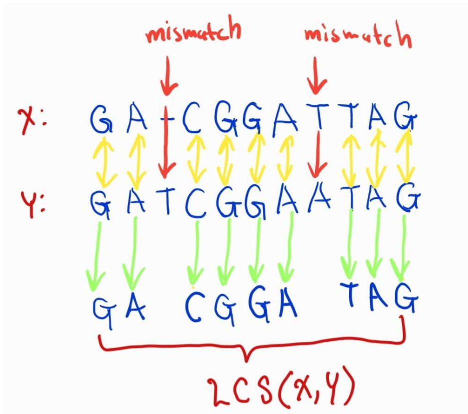
LCS finds its application in DNA sequence alignment

- used to compares similarity between two DNA sequences
- Longest Common Subsequence -> ***Best Alignment***

Longest Common Subsequence: Example

Consider the following DNA fragments *X* and *Y*:

X : GACGGATTAG and *Y* : GATCGGAATAG



Therefore, the longest common subsequence is GACGGATAG.

Longest Common Subsequence: Notation

Notation:

Given two strings X of length m and Y of length n,

$$X = < x_1, x_2, x_3, \dots, x_m >$$
$$Y = < y_1, y_2, y_3, \dots, y_n >$$

$$X_i = < x_1, x_2, x_3, \dots, x_i >$$
$$Y_j = < y_1, y_2, y_3, \dots, y_j >$$

$LCS(X_i, Y_j)$: longest common subsequence of X_i and Y_j

$LCS(X, Y) = LCS(X_m, Y_n)$

$LCS(i, j)$: ***length of*** $LCS(X_i, Y_j)$

Longest Common Subsequence: Optimal Substructure

Theorem (Optimal Substructure):

Let $X =< x_1, x_2, \ldots, x_m >$ and $Y =< y_1, y_2, \ldots, y_n >$ be sequences.

Let $Z =< z_1, z_2, \ldots, z_k >$ be any LCS of X and Y.

1) If $x_m = y_m$, then $z_k = x_m = y_m$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.

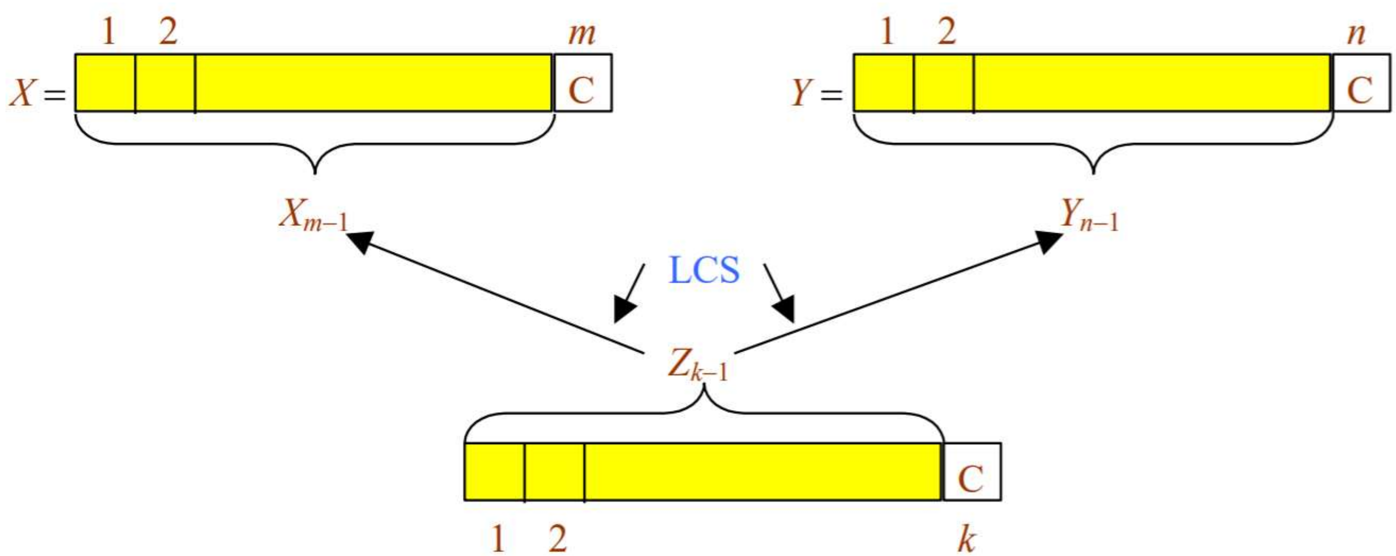
2) If $x_m \neq y_m$, then $z_k \neq x_m$ implies Z is an LCS of X_{m-1} and Y_n.

3) If $x_m \neq y_m$, then $z_k \neq y_n$ implies Z is an LCS of X_m and Y_{n-1}.

Longest Common Subsequence: Optimal Substructure

CASE I:

If $x_m = y_m$, then $z_k = x_m = y_m$ and Z_{k-1} is an LCS of X_{m-1} and Y_{n-1}.



Longest Common Subsequence: Optimal Substructure

Proof: Assume $z_k \neq x_m = y_m$.

We can append $x_m = y_m$ to Z to obtain a subsequence of length $k + 1$, which contradicts optimality of Z.

Therefore, $z_k = x_m = y_m$.

Hence, the prefix Z_{k-1} is a common subsequence (***CS***) of length $k - 1$.

We must show that Z_{k-1} is, in fact, a LCS of X_{m-1} and Y_{n-1}.

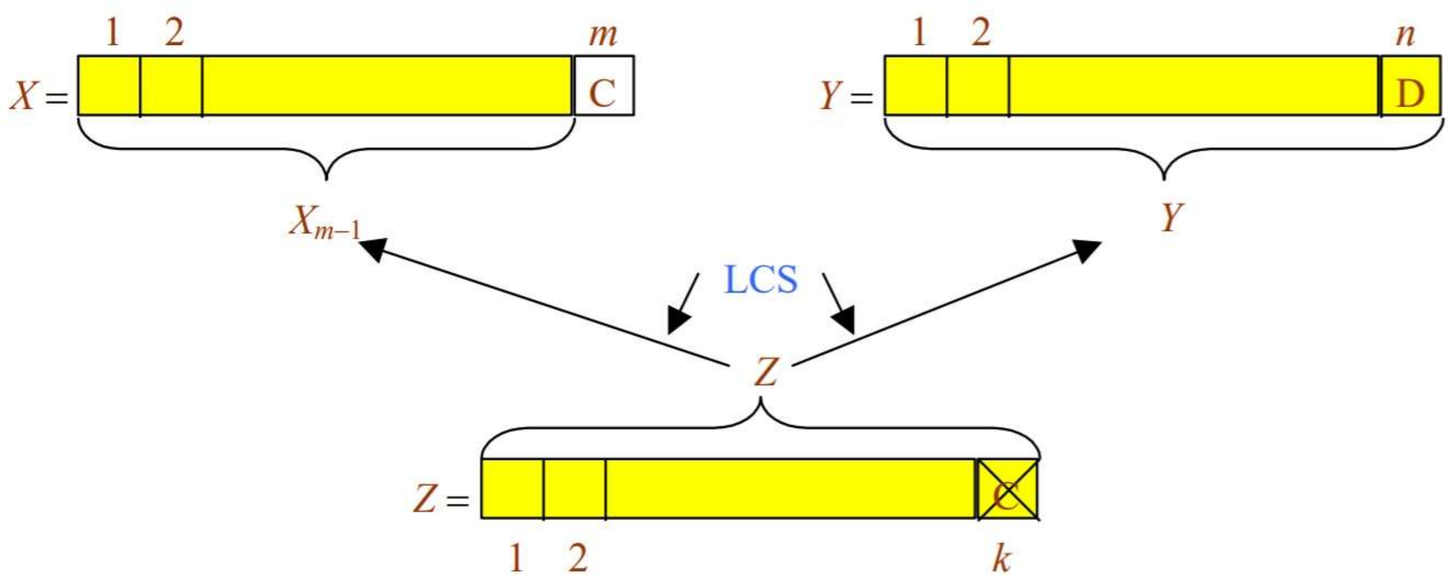
We will prove using a ***cut-and-paste argument*** as follows:

Assume there exists a CS W of X_{m-1} and Y_{n-1} with $|W| = k$.
Appending $x_m = y_n$ to W *wi*ll produce a CS of length $k + 1$, contradicting optimality of Z whose length is k. ■

Longest Common Subsequence: Optimal Substructure

CASE II:

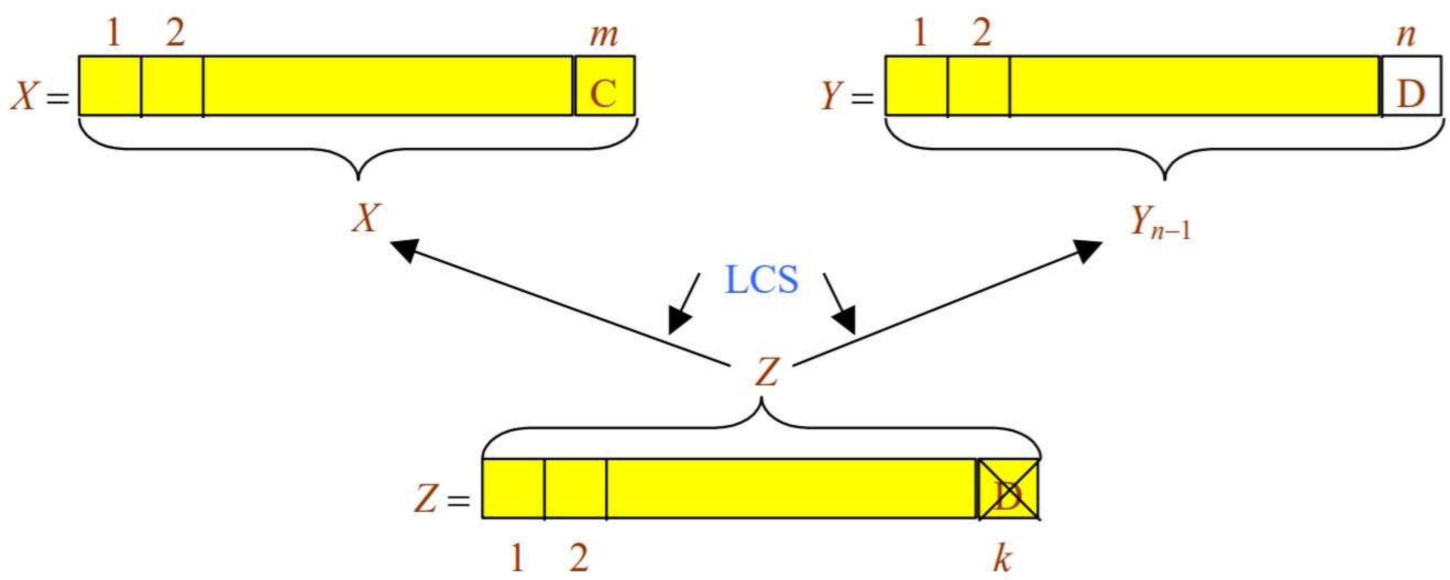
If $x_m \neq y_m$, then $z_k \neq x_m$ implies Z is an LCS of X_{m-1} and Y_n.



Longest Common Subsequence: Optimal Substructure

CASE III:

If $x_m \neq y_m$, then $z_k \neq y_n$ implies Z is an LCS of X_m and Y_{n-1}.



Longest Common Subsequence: Optimal Substructure

Proof: If $z_k \neq x_m$ then Z is a CS of X_{m-1} and Y_n.

We have to show that Z is, in fact, an LCS of X_{m-1} and Y_n.

Assume that there exists a CS W of X_{m-1} and Y_n with $|W| > k$.

Then, W would also be a CS of X_m and Y_n, hence contradicting optimality of Z whose length is k.

Therefore, Z is a LCS of X_{m-1} and Y_n. ■

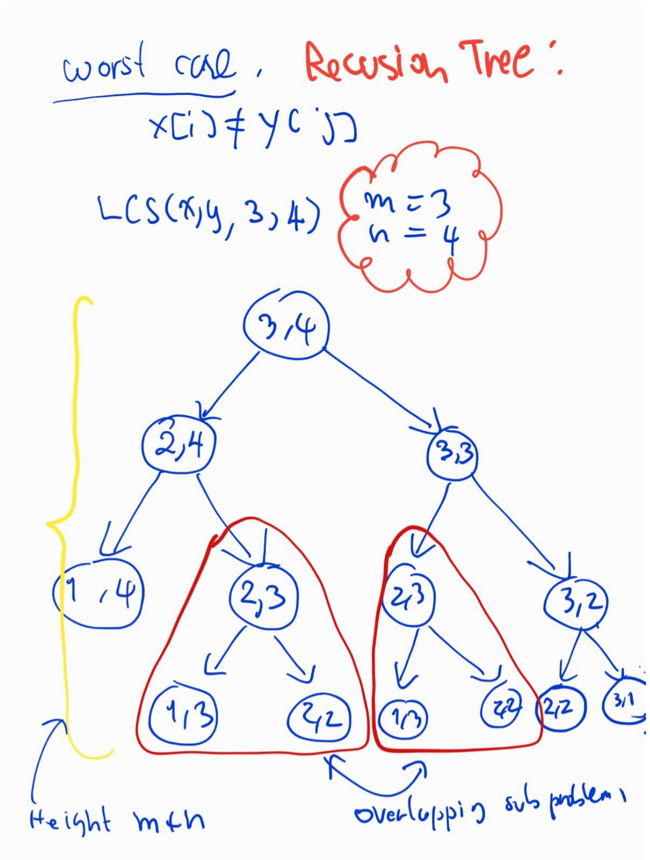
Proof for ***Case III*** is symmetric to the proof of ***Case II***.

Longest Common Subsequence: Recursive Formulation

Recursive Formulation:

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

Longest Common Subsequence: Overlapping Subproblems



Longest Common Subsequence: Overlapping Subproblems

Additionally, the number of ***distinct*** subproblems is relatively ***small*** (i.e. ***polynomial*** in problem size).

The number of distinct subproblems is

$\Theta(mn)$, where m and n are the lengths of X and Y, respectively .

Therefore, we can use ***memoization*** or ***tabulation*** to solve the LCS problem.

Longest Common Subsequence: Bottom-Up

```
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] = "↖"
        elseif c[i − 1, j] ≥ c[i, j − 1]
            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
```

Longest Common Subsequence: Solution Reconstruction

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

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

Summary

We have covered the topic of ***Dynamic Programming*** using

- Fibonacci numbers
- Matrix Chain Multiplication (MCM)
- Longest Common Subsequence (LCS)

as examples.

Central to DP, are ***optimal substructure*** and ***overlapping subproblem*** properties.

We will cover ***Greedy Algorithms*** next week.