Algorithms

Kiho Choi

Fall, 2022

Department of AI·Software
Gachon University

3. Dynamic Programming I

Contents

Dynamic programming

- Dynamic programming is a design technique for solving optimization problems having a recursive structure.
- Dynamic programming is related partially to Divide and Conquer and to Greedy Algorithms.
- Problem at hand is subdivided to partial problems (subproblems) whose results are combined to solve the overall problem

Strategy

- Big Idea: avoid recomputation
- General strategy is to solve all subproblems once and only once, to store all the subsolutions in a table, and to reuse optimal subproblem solutions
- Typically, the total number of distinct subproblems is a polynomial in the input size

Four-step method

- 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.

Elements of dynamic programming

- Optimal substructure: an optimal solution to the problem contains within it optimal solutions to subproblems.
- Overlapping substructure: a recursive algorithm revisits the same problem over and over again

The Fibonacci Sequence

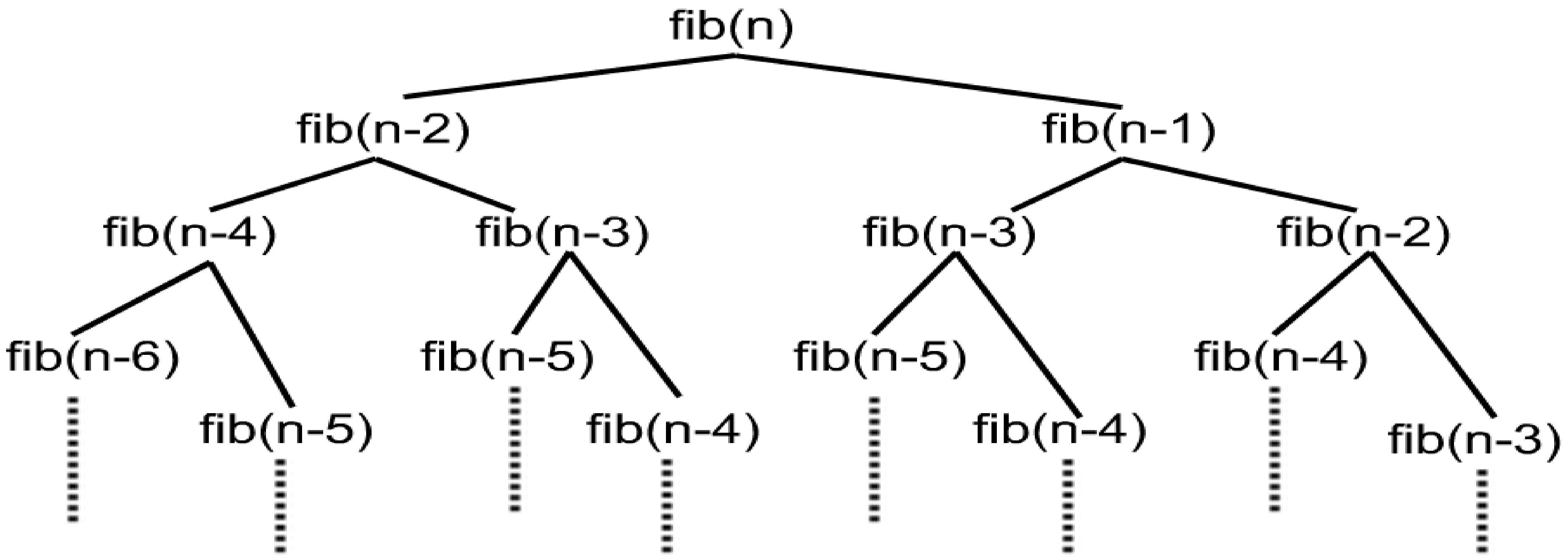
Example : The fibonacci sequence

$$
\begin{aligned}
fib(0) &= 0 \\
fib(1) &= 1 \\
fib(2) &= 1 \\
fib(3) &= 2 \\
fib(4) &= 3 \\
fib(5) &= 5 \\
&\ldots \\
fib(n) &= fib(n-1) + fib(n-2)
\end{aligned}
$$

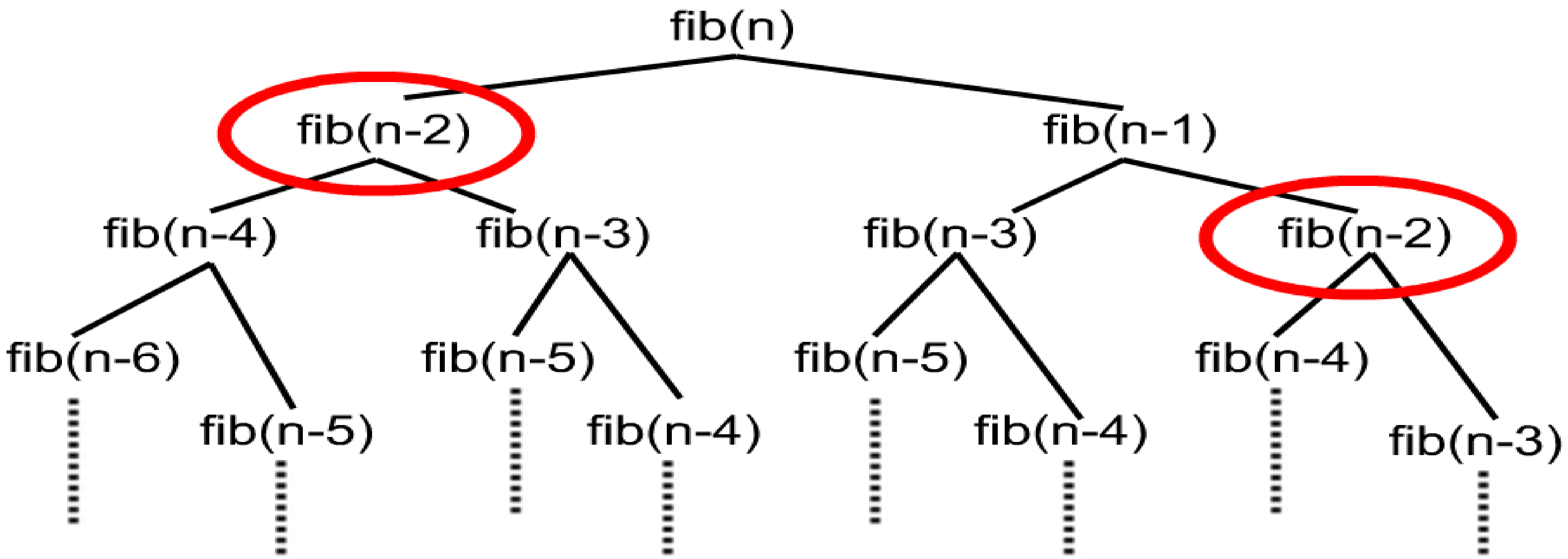
Naïvely computing fib(n)

```
int fib(int n) {
    if (n == 0 || n == 1) {
        return n;
    } else {
        return (fib(n-1) + fib(n-2));
    }
}
```

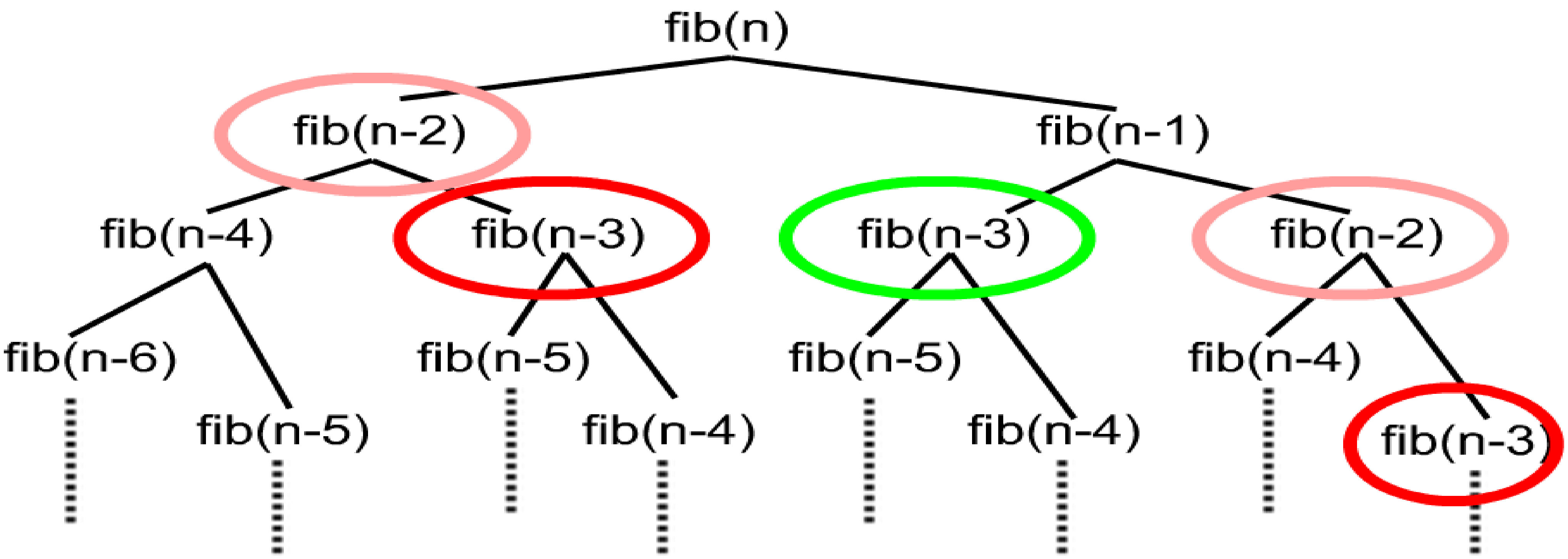
Naïvely computing fib(n)



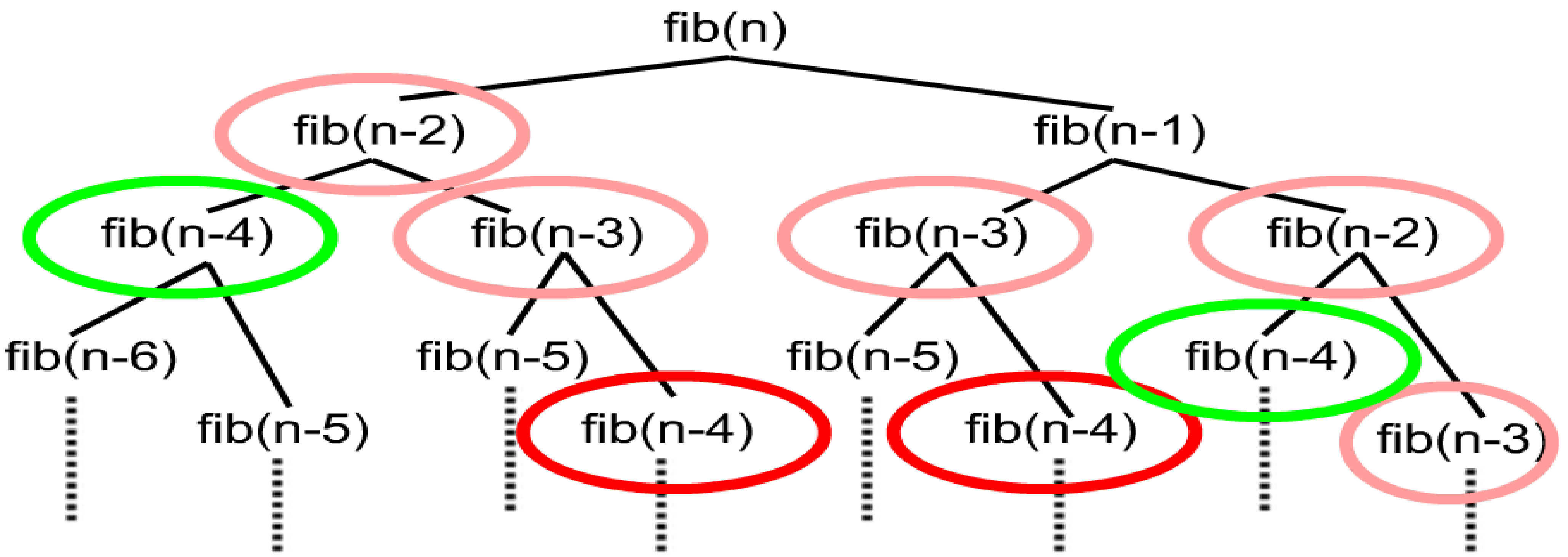
Naïvely computing fib(n)



Naïvely computing fib(n)



Naïvely computing fib(n)



How to fix things…

- The solution to fib(n) depends on:
  - The solution to fib(n - 1).
  - The solution to fib(n - 2).
- Use dynamic programming to organize everything:
  - Each subproblem is characterized by the value of its input.
  - To compute fib(n), we need fib(n - 2) and fib(n - 1).
  - So compute fib(0) first, and work upwards.
  - Use an array to keep track of everything.

Computing fib(n) : Take 2

```
int fib(int n) {
    int[] seq = new int[n+1];
    seq[0] = 0;
    seq[1] = 1;
    for (int i = 2; i <= n; i++) {
        // looks like fib(i) = fib(i - 1) + fib(i - 2)
        seq[i] = seq[i - 1] + seq[i - 2];
    }
    return seq[n];
}
```

Computing fib(n) : Take 2

```
int fib(int n) {

    // This array will keep track of the solutions to
    // all n+1 problems.
    int[] seq = new int[n + 1];

    ...
```

Computing fib(n) : Take 2

```
int fib(int n) {
    ...
    // Assume that (n >= 1).  Otherwise, seq is to small.
    // The smallest subproblems have trivial solutions.
    seq[0] = 0;
    seq[1] = 1;
    ...
```

Computing fib(n) : Take 2

```
int fib(int n) {
    ...
    // Solve other subproblems in order of increasing size.
    // Use the fact that we've already solved even smaller
    // subproblems already.
    for (int i = 2; i <= n; i++) {
        seq[i] = seq[i - 1] + seq[i - 2];
    }
    ...
```

Computing fib(n) : Take 2

```
int fib(int n) {

    ...

    // Finally, return solution to the actual problem.
    return seq[n];
}
```

Longest common subsequence

Design technique, like divide-and-conquer.

Example: *Longest Common Subsequence (LCS)*

- Given two sequences $x[1 \,..\, m]$ and $y[1 \,..\, n]$, find a longest subsequence common to them both.

“a” *not* “the”

x: A B C B D A B

y: B D C A B A

BCBA = LCS(x, y)

functional notation, but not a function

Longest common subsequence

Optimal substructure

An optimal solution to a problem (instance) contains optimal solutions to subproblems.

If $z = \mathrm{LCS}(x, y)$, then any prefix of z is an LCS of a prefix of x and a prefix of y.

Longest common subsequence

$$
\begin{array}{l}
\mathrm{LCS}(x, y, i, j) \\
\quad \textbf{if } x[i] = y[j] \\
\qquad \textbf{then } c[i, j] \leftarrow \mathrm{LCS}(x, y, i-1, j-1) + 1 \\
\qquad \textbf{else } c[i, j] \leftarrow \max\{\mathrm{LCS}(x, y, i-1, j), \\
\qquad\qquad\qquad\qquad\qquad \mathrm{LCS}(x, y, i, j-1)\}
\end{array}
$$

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

Longest common subsequence

Overlapping subproblems

A recursive solution contains a "small" number of distinct subproblems repeated many times.

The number of distinct LCS subproblems for two strings of lengths m and n is only mn.

Longest common subsequence

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

$$
\begin{array}{l}
\mathrm{LCS}(x, y, i, j) \\
\quad \textbf{if } c[i, j] = \mathrm{NIL} \\
\quad\quad \textbf{then if } x[i] = y[j] \\
\quad\quad\quad \textbf{then } c[i, j] \leftarrow \mathrm{LCS}(x, y, i-1, j-1) + 1 \\
\quad\quad\quad \textbf{else } c[i, j] \leftarrow \max\{\mathrm{LCS}(x, y, i-1, j), \\
\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad\quad \mathrm{LCS}(x, y, i, j-1)\}
\end{array}
\left.\vphantom{\begin{array}{l}a\\a\\a\\a\end{array}}\right\} \textit{same as before}
$$

Longest common subsequence

Memoization: After computing a solution to a subproblem, store it in a table. Subsequent calls check the table to avoid redoing work.

LCS(x, y, i, j)
    if $c[i, j] = \text{NIL}$
        then if $x[i] = y[j]$
            then $c[i, j] \leftarrow \text{LCS}(x, y, i-1, j-1) + 1$
            else $c[i, j] \leftarrow \max\{\text{LCS}(x, y, i-1, j), \text{LCS}(x, y, i, j-1)\}$

same as before

Time $= \Theta(mn) =$ constant work per table entry.
Space $= \Theta(mn)$.

Longest common subsequence

IDEA:
Compute the table bottom-up.

Time $= \Theta(mn)$.

Reconstruct LCS by tracing backwards.

Space $= \Theta(mn)$.
Exercise:
$O(\min\{m, n\})$.

		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

Implementation

Implementation of Longest common subsequence

$$
\begin{aligned}
&\mathrm{LCS}(x, y, i, j) \\
&\quad \textbf{if } x[i] = y[j] \\
&\qquad \textbf{then } c[i, j] \leftarrow \mathrm{LCS}(x, y, i-1, j-1) + 1 \\
&\qquad \textbf{else } c[i, j] \leftarrow \max\left\{\mathrm{LCS}(x, y, i-1, j),\ \mathrm{LCS}(x, y, i, j-1)\right\}
\end{aligned}
$$

```python
def lcs(X, Y, m, n):
    if m == 0 or n == 0:
       return 0;
    elif X[m-1] == Y[n-1]:
       return 1 + lcs(X, Y, m-1, n-1);
    else:
       return max(lcs(X, Y, m, n-1), lcs(X, Y, m-1, n));


# Input texts
X = "ABCBDAB"
Y = "BDCABA"

# Print results
print("Length of LCS is ", lcs(X, Y, len(X), len(Y)))
```

```
Length of LCS is  4
```

Implementation of Longest common subsequence

		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

```python
def lcs(X, Y):
    # Length
    m = len(X)
    n = len(Y)

    # Array
    L = [[None]*(n + 1) for i in range(m + 1)]

    # Filling array from the bottom
    for i in range(m + 1):
        for j in range(n + 1):
            if i == 0 or j == 0 :
                L[i][j] = 0
            elif X[i-1] == Y[j-1]:
                L[i][j] = L[i-1][j-1]+1
            else:
                L[i][j] = max(L[i-1][j], L[i][j-1])
    return L[m][n]

# Input texts
X = "ABCBDAB"
Y = "BDCABA"

# Print results
print("Length of LCS is ", lcs(X, Y))
```

```
Length of LCS is  4
```

Example code test

- Code test: https://www.acmicpc.net/problem/9251
- Solving the problem using Longest common subsequence
- Example result of submission

9251	aikiho	모든 언어	모든 결과	검색

제출 번호	아이디	문제	결과	메모리	시간	언어	코드 길이	제출한 시간
48920370	aikiho	9251	맞았습니다!!	55032 KB	732 ms	Python 3 / 수정	404 B	1분 전

THANK YOU