**SUBJECT**: DESIGN AND ANALYSIS OF ALGORITHMS

**CODE**: 503040

Duration: 150 minutes

Allowed to use materials.

**LAB 07: Dynamic Programming (Part 2)**

# Objectives

Understand the properties of Dynamic Programming algorithm design technique

Be able to design, implement, and analyze Dynamic Programming algorithms solving common problems.

# Idea

- set up a recurrence relating a solution to a larger instance to solutions of some smaller instances

- solve smaller instances once

- record solutions in a table

- extract solution to the initial instance from that table

## An example of a Dynamic Programming algorithm

Implement and analyze a dynamic programming algorithm to compute n-th Fibonacci number

The implementation in Python is presented as follows

|  |  |  |
| --- | --- | --- |
| |  |  | | --- | --- | | 1  2  3  4  5  6  7  8  9  10  11  12  13  14  15  16 | **def** **fibonacci**(n):  """  calculates n-th Fibonacci numbers  input:  n(int) - the ordinal number >= 0  output:  F (int) - n-th Fibonacci number  """  F = [**0** **for** \_ **in** range(n+**1**)]  F[**0**] = **0**  F[**1**] = **1**  **for** i **in** range(**2**,n+**1**):  F[i] = F[i-**1**] + F[i-**2**]  **return** F[n]  **print**(fibonacci(**4**)) | |

Analysis:

1/ Basic operation: addition on line 13

2/ Worst case: as average case

3/Counting the number of basic operations in the worst case:

…

**Time efficiency**

***T*(*n*) = *n*-1 ∈ Θ(*n*)**

(\*\*\*) How to draw running time graphics

|  |
| --- |
| def fac(n):  """  ...  """  if n == 0:  return 1  result = 1  for i in range(1, n):  result \*= i  return result |
| import time  def measure\_time(func, N):  """  ...  """  runtime = []  for n in N:  start = time.time()  f = func(n)  stop = time.time()  runtime.append(stop-start)  return runtime |
| import pylab |
| N = list(range(100))  rtime = measure\_time(fac, N)  rtime2 = [t\*1.5 for t in rtime] |
| pylab.plot(N, rtime, N, rtime2)  pylab.legend(['1', '2']) |
|  |

# Exercises

**Warm up**

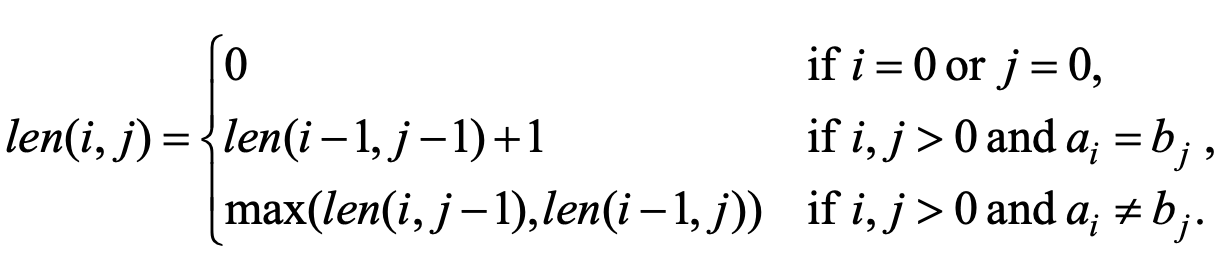
1. Computing Longest Common Subsequence (LCS).
2. Implement (in Python) and analyze a dynamic programming (DP) algorithm to solve the problem.

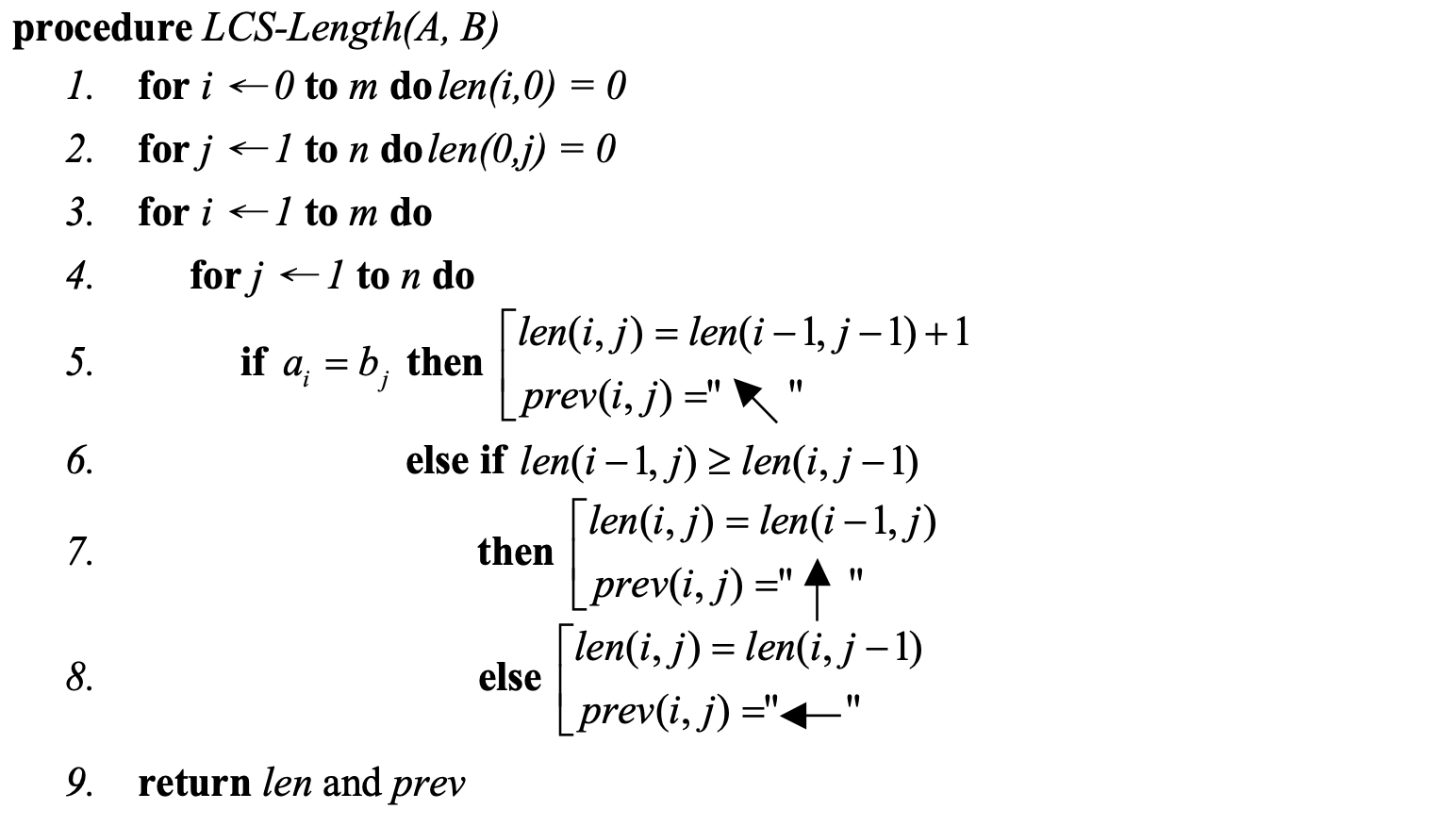
Hint:

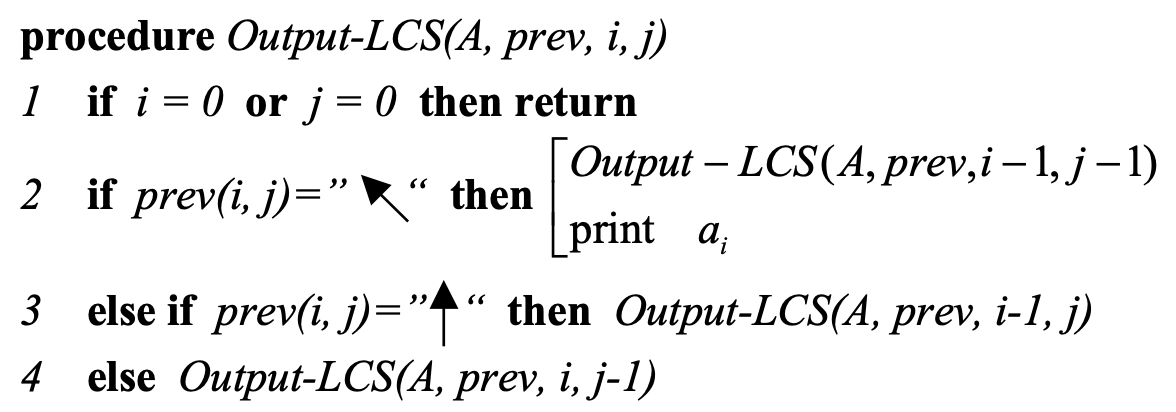
Let A*=a1a2…am* and *B=b1b2…bn .*

*len*(*i, j*): the length of an LCS between   
 *a1a2…ai* and *b1b2…bj*

With proper initializations, *len*(*i, j*) can be computed as follows.

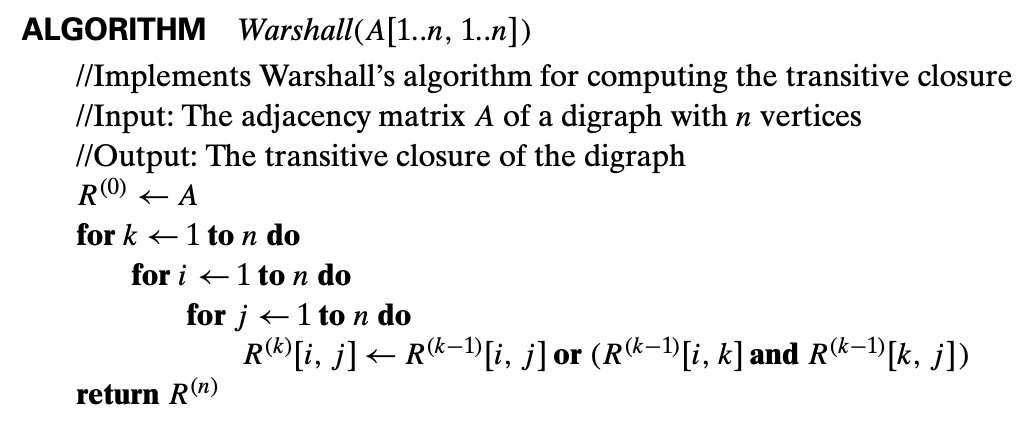




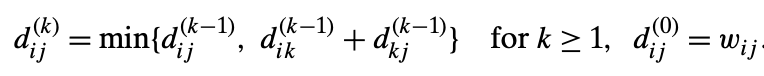
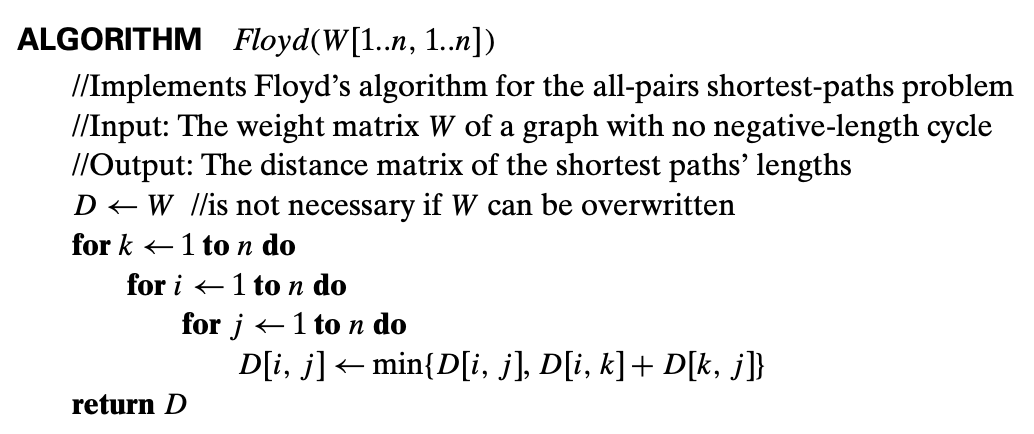


**Intermediate exercises**

1. Computing the transitive closure of a relation (Alternatively: existence of all nontrivial paths in a digraph).
2. Implement (in Python) and analyze a dynamic programming (DP) algorithm to solve the problem.
3. In the same axes, draw the graphics of the actual running time of the DP program and a divide-and-conquer solution.



1. All pairs shortest path distances.
2. Implement (in Python) and analyze a dynamic programming (DP) algorithm to solve the problem.
3. In the same axes, draw the graphics of the actual running time of the DP program and a divide-and-conquer solution.

**Upper-Intermediate exercise**

1. Revising Floyd algorithm to print out the shortest path which starts at a vertex  and ends at another vertex . Implement (in Python) a dynamic programming (DP) algorithm to solve the problem.

Hint: create matrix P to store shortest paths, where the value of the element P[i][j] is the index of a next vertex in the shortest path which starts at vertex i and ends at vertex j.