Recursions and Dynamic Programming

Il-Chul Moon
Dept. of Industrial and Systems Engineering
KAIST
icmoon@kaist.ac.kr

Weekly Objectives

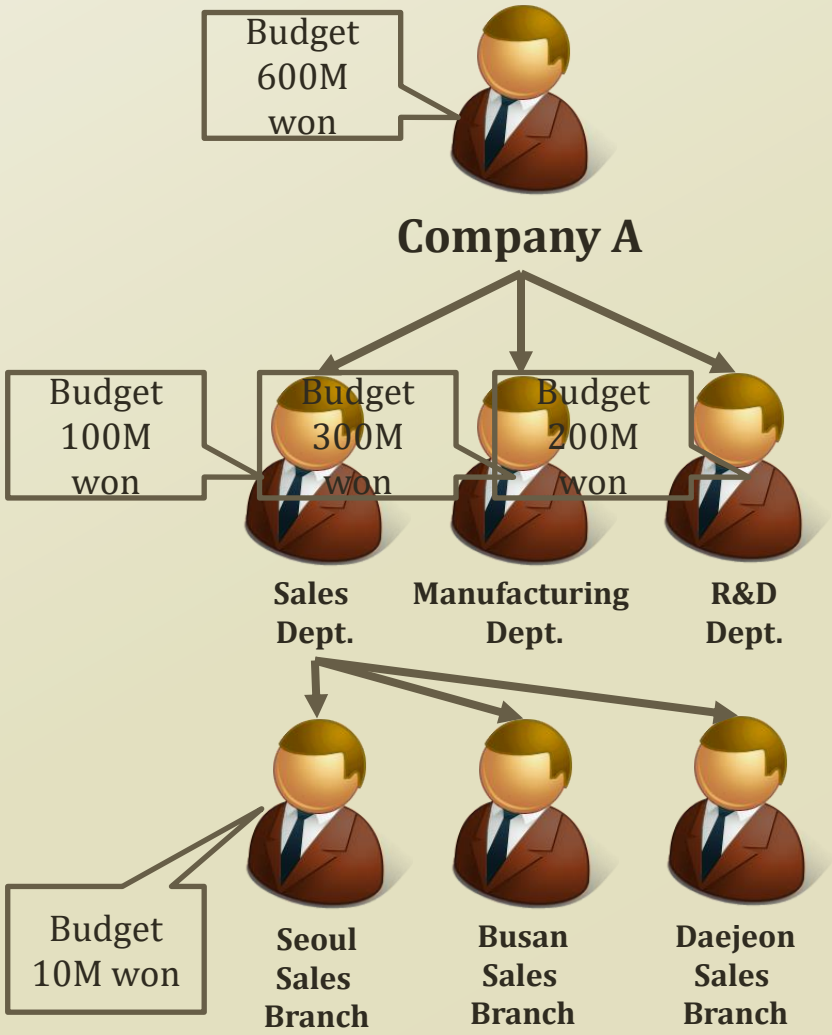
- This week, we learn how to program recursive routines and dynamic programming concepts
  - Recursion
  - Dynamic programming
- Objectives are
  - Understanding the concept of recursions
    - Repeating problems
    - Divide and conquer
    - Recursion function call
    - Recursion escape
    - Recursion depth
  - Able to implement recursive programs
  - Understanding the concept of dynamic programming
    - Reusing previous function call result
    - Memoization for time saving

RECURSION

Repeating Problems and Divide and Conquer

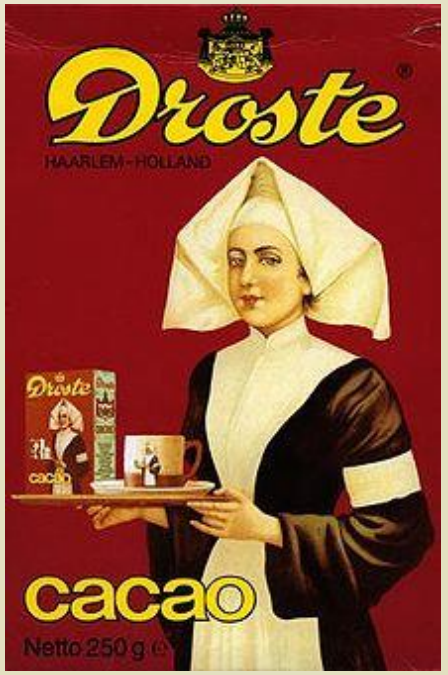
- Calculating a budget of a company?
  - Departments consist of the company
  - Departments within departments
- Can't avoid the below structures
  - class Department
    - dept = [sales, manu, randd]
    - def **calculateBudget**(self)
      - Sum = 0
      - For itr in range(0, numDepartments)
        - Sum = sum + dept[itr].**calculateBudget**()
      - Return sum

Russian Doll:
Dolls within dolls



More examples…

- Factorial
  - $Factorial(n) = \begin{cases} 1 & if\ n = 0 \\ n \times (n-1) \times \cdots \times 2 \times 1 & if\ n > 0 \end{cases}$
  - Repeating problems?
    - $Factorial(n) = \begin{cases} 1 & if\ n = 0 \\ n \times Factorial(n-1) & if\ n > 0 \end{cases}$
- Great Common Divisor
  - GCD(32,24) = 8
  - Euclid's algorithm
    - GCD(A, B)=GCD(B, A mod B)
    - GCD(A, 0)=A
- Commonality
  - Repeating function calls
  - Reducing parameters
  - Just like the mathematical induction



Self-Similar

Recursion

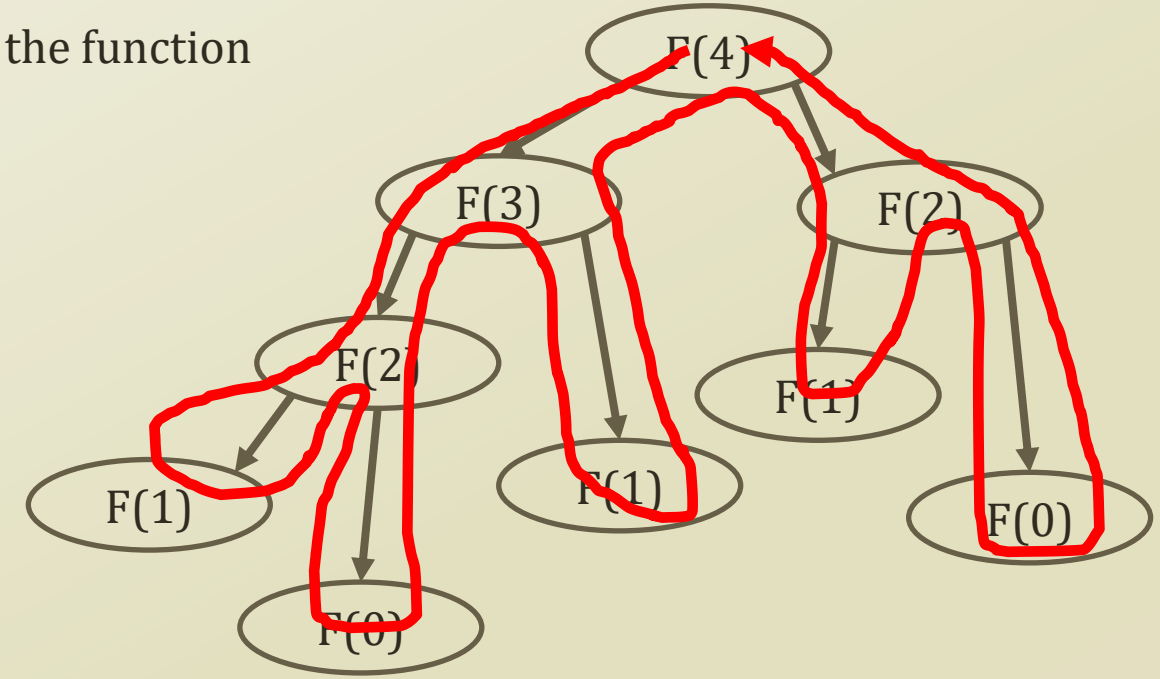
- A programming method to handle the repeating items in a self-similar way
  - Often in a form of
    - Calling a function within the function
      - def functionA(target)
        - ….
        - functionA(target')
        - ….
        - if (escapeCondition)
          - Return A;

```
def Fibonacci(n):
    if n == 0:
        return 0
    if n == 1:
        return 1
    intRet = Fibonacci(n-1) + Fibonacci(n-2)
    return intRet

for itr in range(0, 10):
    print(Fibonacci(itr), end=" ")
```

```
0 1 1 2 3 5 8 13 21 34
```

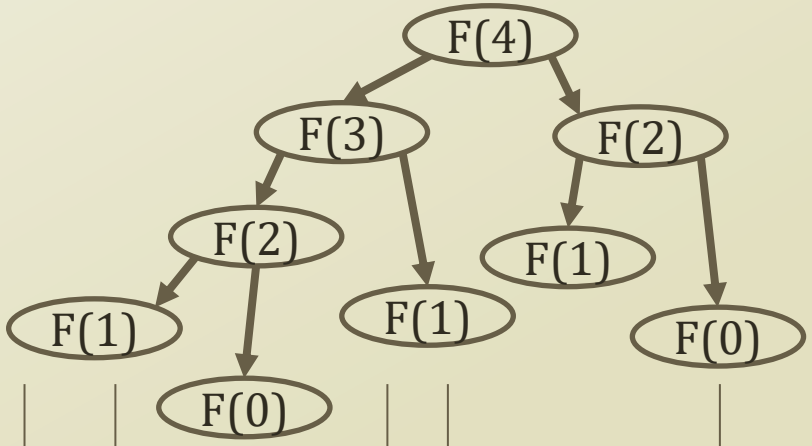
Program Execution Flow

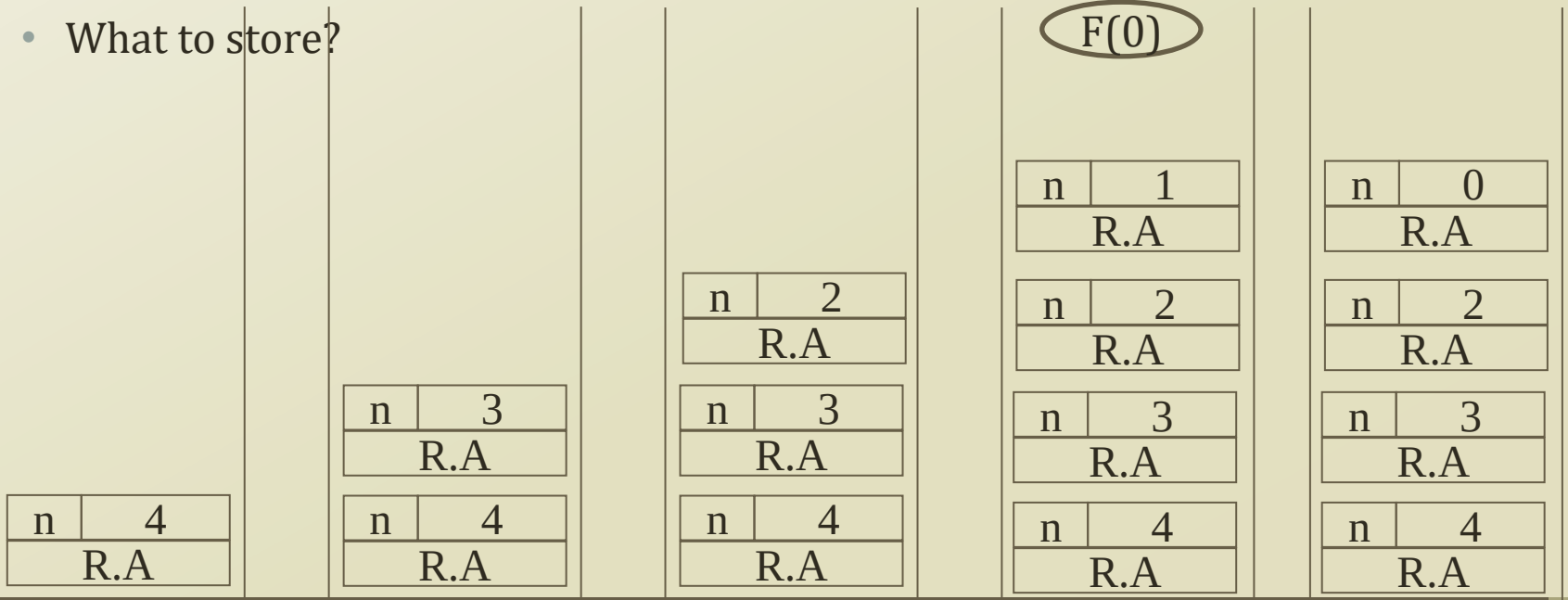


$$Fibonacci(n) = \begin{cases} 0 & n = 0 \\ 1 & n = 1 \\ Fibonacci(n-1) + Fibonacci(n-2) & n \geq 2 \end{cases}$$

Recursions and Stackframe

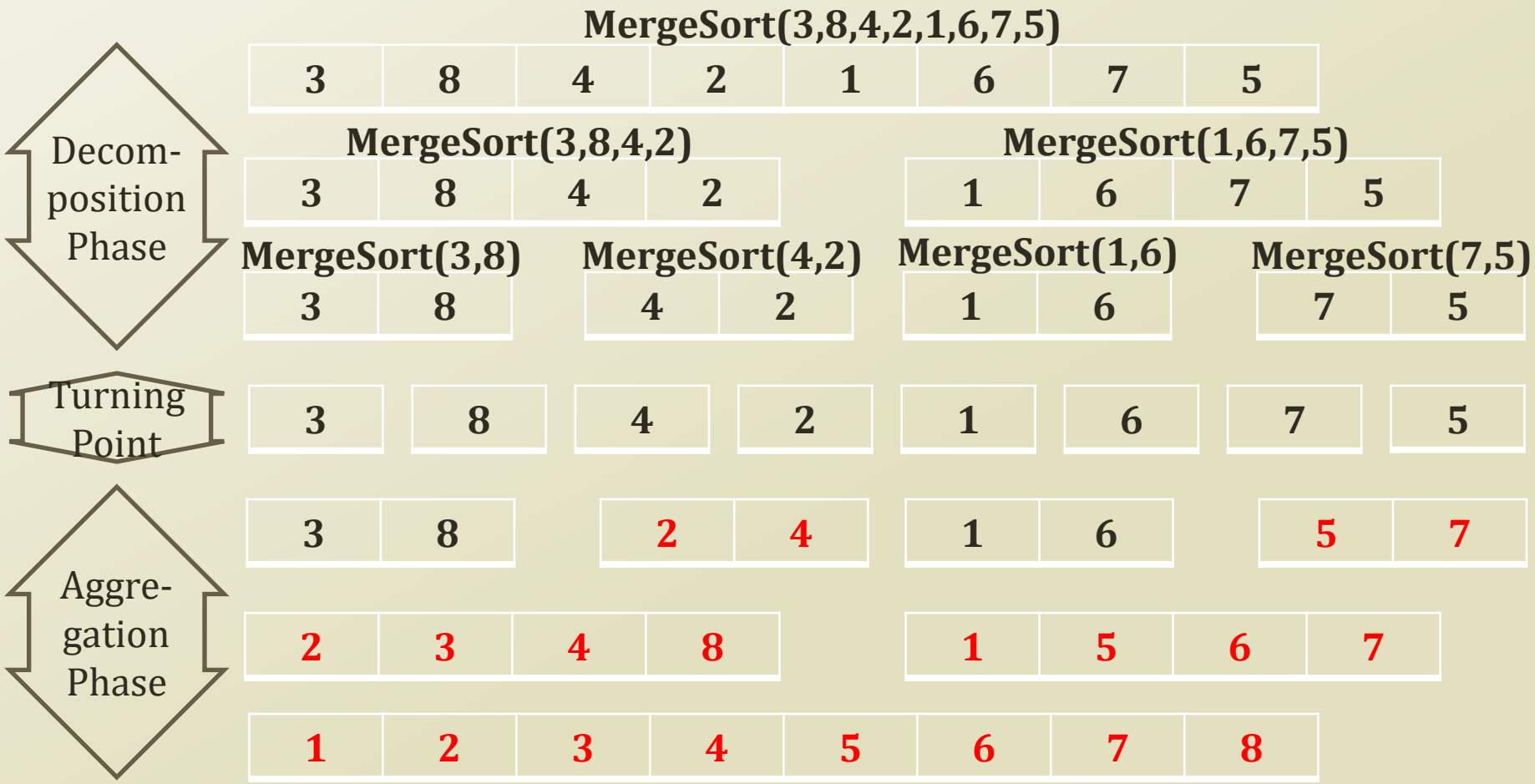
- Recursion of functions
  - Increase the items in the stackframe
    - Stackframe is a stack storing your function call history
      - Push: When a function is invoked
      - Pop: When a function hits *return* or ends
      - What to store?
        - Local variables and function call parameters





Merge Sort

- Merge sort: One example of recursive programming
  - Decompose into two smaller lists
  - Aggregate to one larger and sorted list



Implementation Example: Merge Sort

```python
import random

def performMergeSort(lstElementToSort):
    if len(lstElementToSort) == 1:
        return lstElementToSort

    lstSubElementToSort1 = []
    lstSubElementToSort2 = []
    for itr in range(len(lstElementToSort)):
        if len(lstElementToSort)/2 > itr:
            lstSubElementToSort1.append(lstElementToSort[itr])
        else:
            lstSubElementToSort2.append(lstElementToSort[itr])

    lstSubElementToSort1 = performMergeSort(lstSubElementToSort1)
    lstSubElementToSort2 = performMergeSort(lstSubElementToSort2)

    idxCount1 = 0
    idxCount2 = 0
    for itr in range(len(lstElementToSort)):
        if idxCount1 == len(lstSubElementToSort1):
            lstElementToSort[itr] = lstSubElementToSort2[idxCount2]
            idxCount2 = idxCount2 + 1
        elif idxCount2 == len(lstSubElementToSort2):
            lstElementToSort[itr] = lstSubElementToSort1[idxCount1]
            idxCount1 = idxCount1 + 1
        elif lstSubElementToSort1[idxCount1] > lstSubElementToSort2[idxCount2]:
            lstElementToSort[itr] = lstSubElementToSort2[idxCount2]
            idxCount2 = idxCount2 + 1
        else:
            lstElementToSort[itr] = lstSubElementToSort1[idxCount1]
            idxCount1 = idxCount1 + 1
    return lstElementToSort
```

Execution Code

```python
lstRandom = []
for itr in range(0, 10):
    lstRandom.append( random.randrange(0, 100))
print(lstRandom)
lstRandom = performMergeSort(lstRandom)
print(lstRandom)
```

Decomposition

Recursion

Aggregation

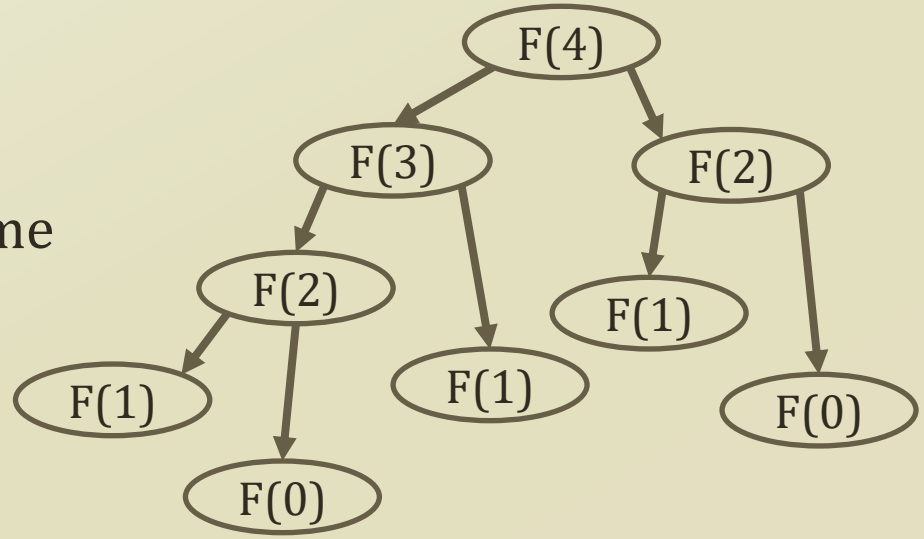
- Code execution timing!
  - Before Recursion

  = Before Branching out
  - After Recursion

  = After Branching out

Problems in Recursions of Fibonacci Sequence

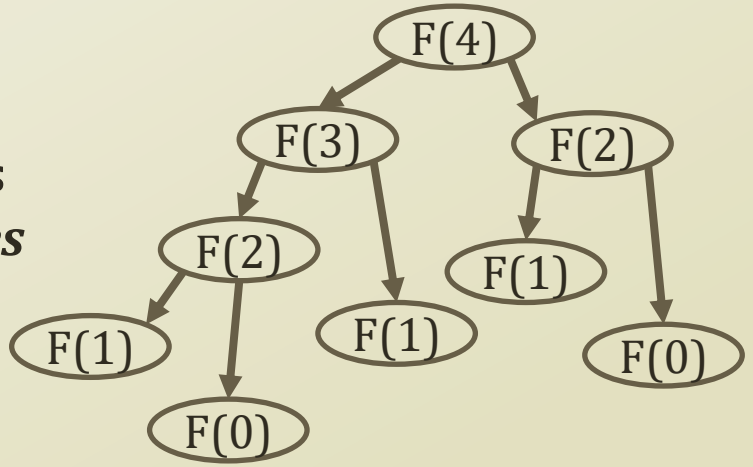
- Problems in recursions
  - Excessive function calls
    - Calling functions again and again
    - Even though the function is executed before with the same parameters
- For instance, Fibonacci(4)
  - Has two repeated calls of F(0)
  - Has three repeated calls of F(1)
  - Has two repeated calls of F(2)
- These are unnecessarily taking time and space
- How to solve this problem?



DYNAMIC PROGRAMMING

Dynamic Programming

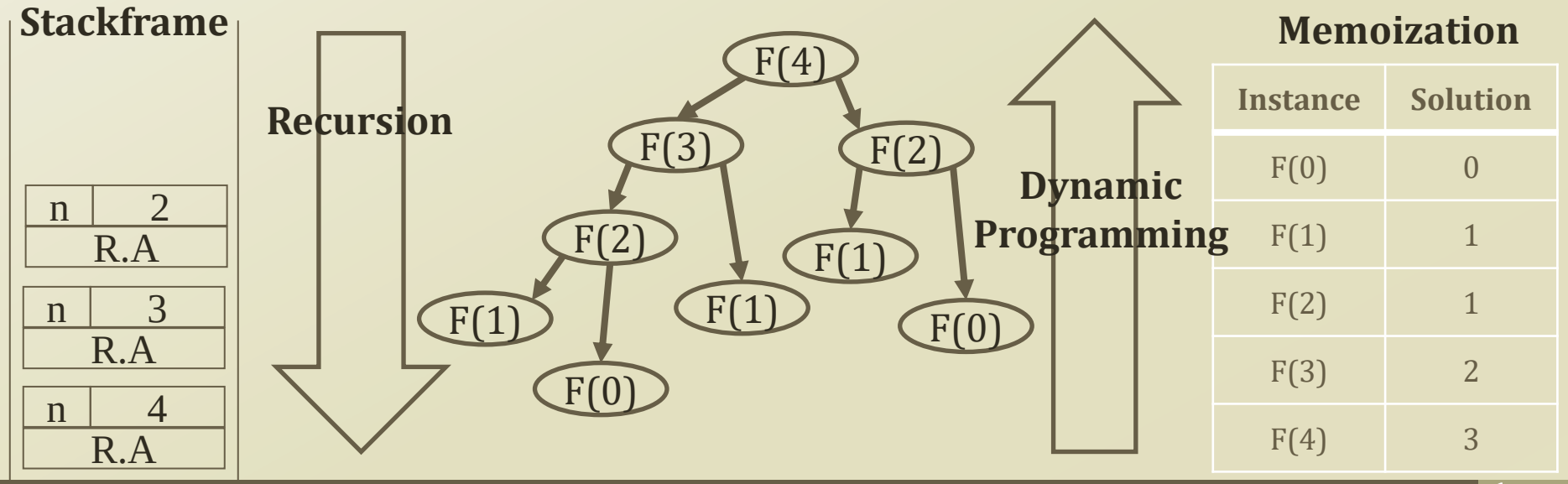
- Dynamic programming:
  - A general algorithm design technique for solving problems defined by or formulated as ***recurrences with overlapping sub-instances***
  - In this context, Programming == Planning
- Main storyline
  - Setting up a recurrence
    - Relating a solution of a larger instance to solutions of some smaller instances
    - Solve small instances once
    - Record solutions in a table
    - Extract a solution of a larger instance from the table



Instance	Solution
F(0)	0
F(1)	1
F(2)	1
F(3)	2
F(4)	?

Memoization

- Key technique of dynamic programming
  - Simply put
    - Storing the results of previous function calls to reuse the results again in the future
  - More philosophical sense
    - Bottom-up approach for problem-solving
      - Recursion: Top-down of divide and conquer
      - Dynamic programming: Bottom-up of storing and building



Implementation Example: Fibonacci Sequence in DP

```python
def FibonacciDP(n):
    dicFibonacci = {}
    dicFibonacci[0] = 0
    dicFibonacci[1] = 1
    for itr in range(2, n + 1):
        dicFibonacci[itr] = dicFibonacci[itr-1] + dicFibonacci[itr-2]
    return dicFibonacci[n]

for itr in range(0, 10):
    print(FibonacciDP(itr), end=" ")
```

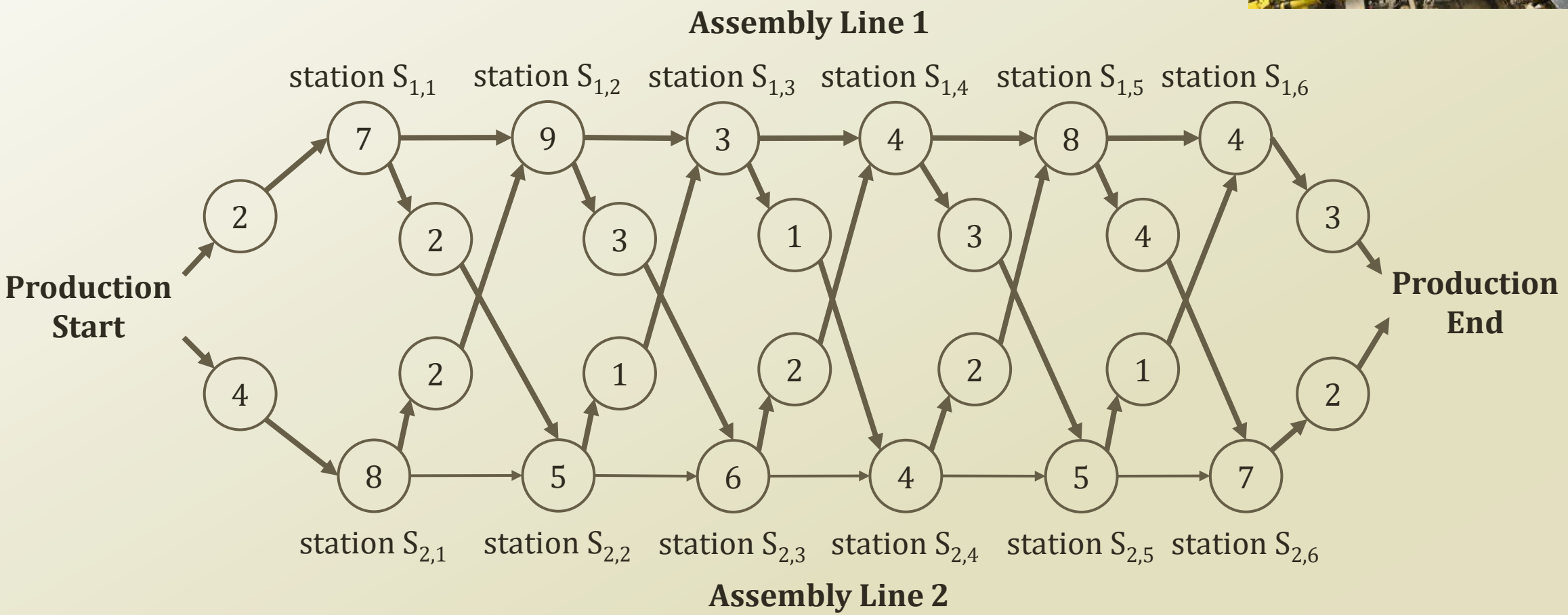
Setting up a memoization table

Building up a bigger solutions

Execution Part

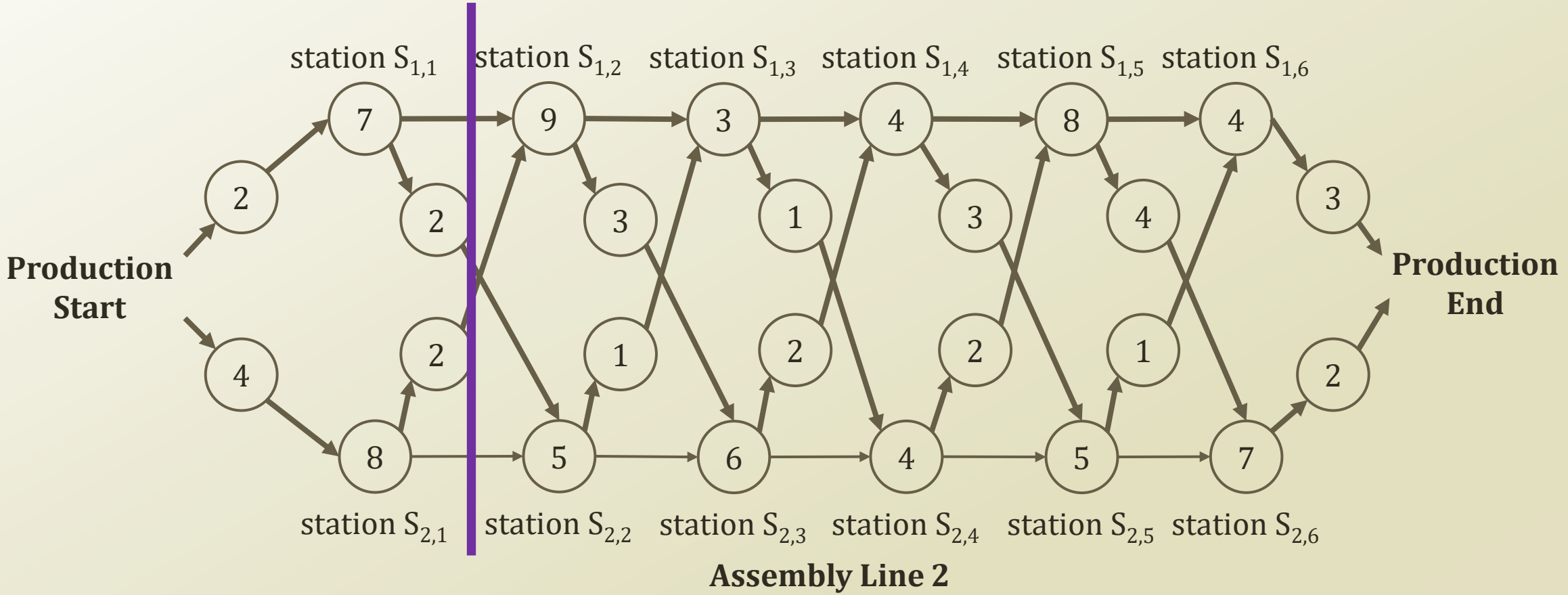
- Use a dictionary collection variable type for memoization
  - Memoization
    - Storing a fibonacci number for a particular index
- Now,
  - We have a new space requirement, the dictionary or the table, of O(N)
  - We have reduced execution time from $O(2^n)$ to O(N)

Assembly Line Scheduling



Goal: Computing the fastest production route

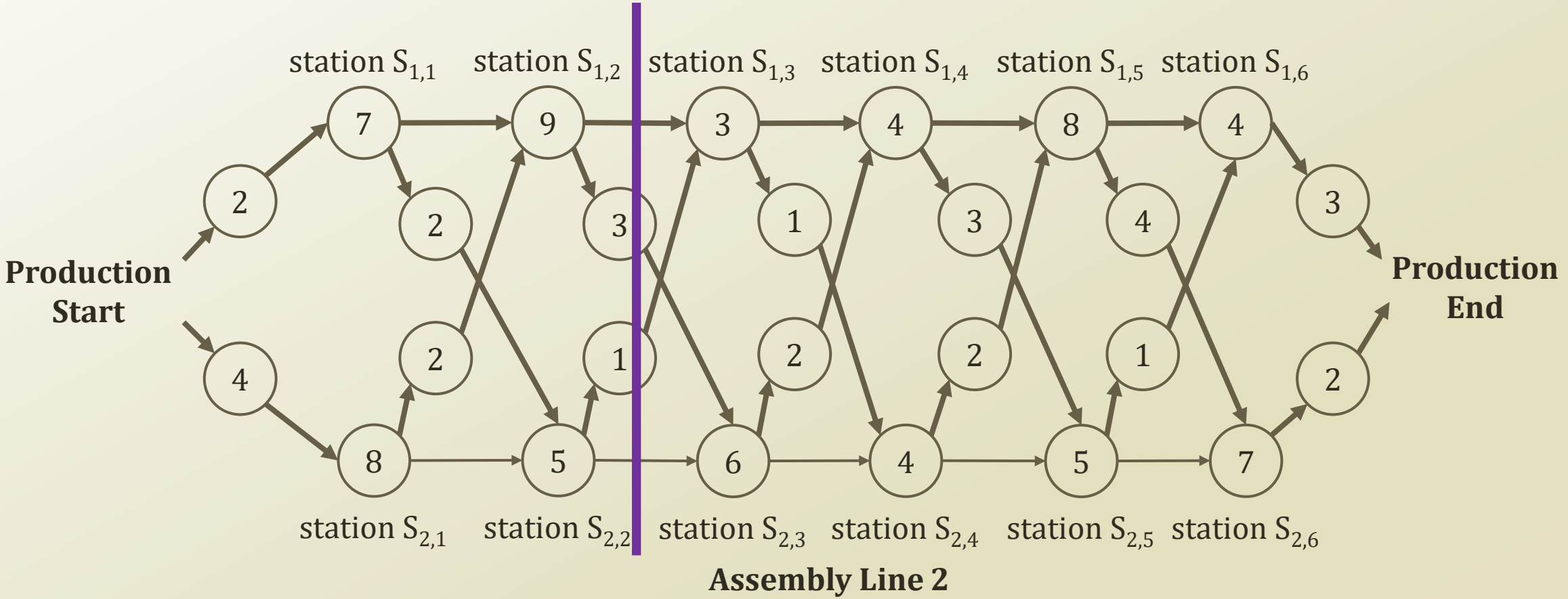
Process of Assembly Line Scheduling



Time	1	2	3	4	5	6
L1	9					
L2	12					

Trace	1	2	3	4	5	6
L1	S					
L2	S					

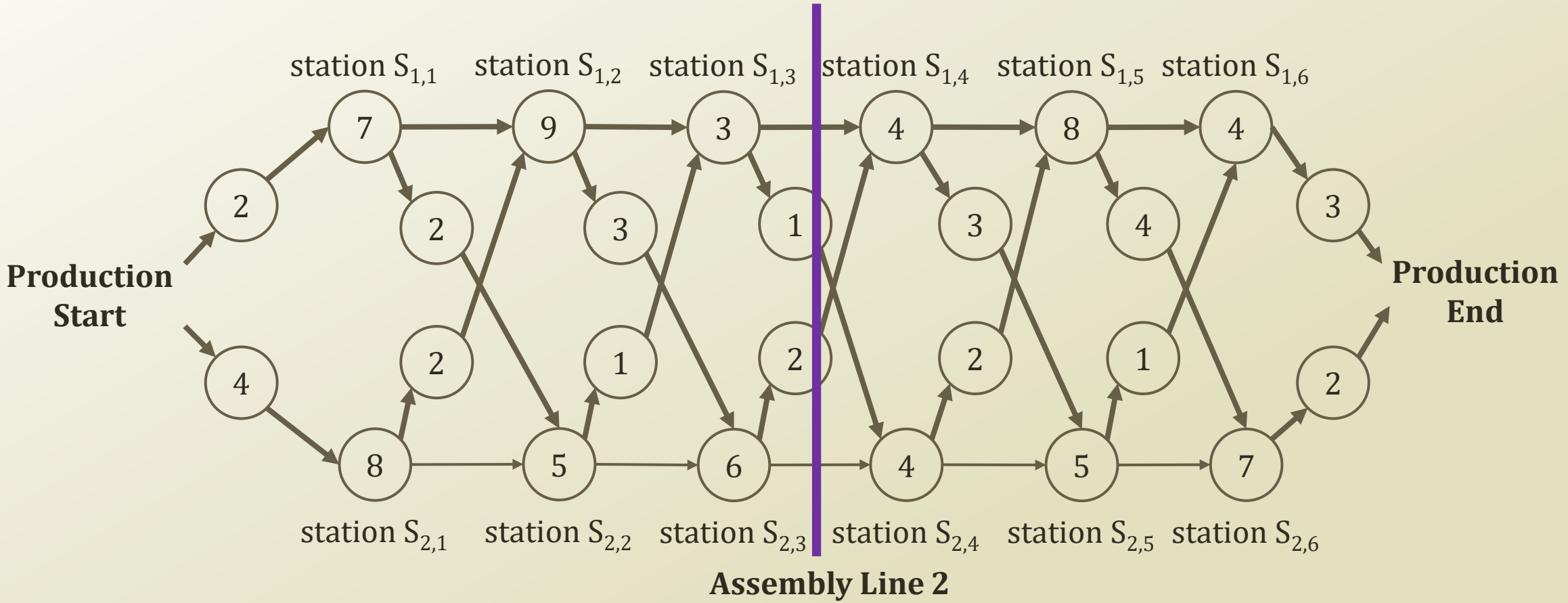
Process of Assembly Line Scheduling



Time	1	2	3	4	5	6
L1	9	18				
L2	12	16				

Trace	1	2	3	4	5	6
L1	S	1				
L2	S	1				

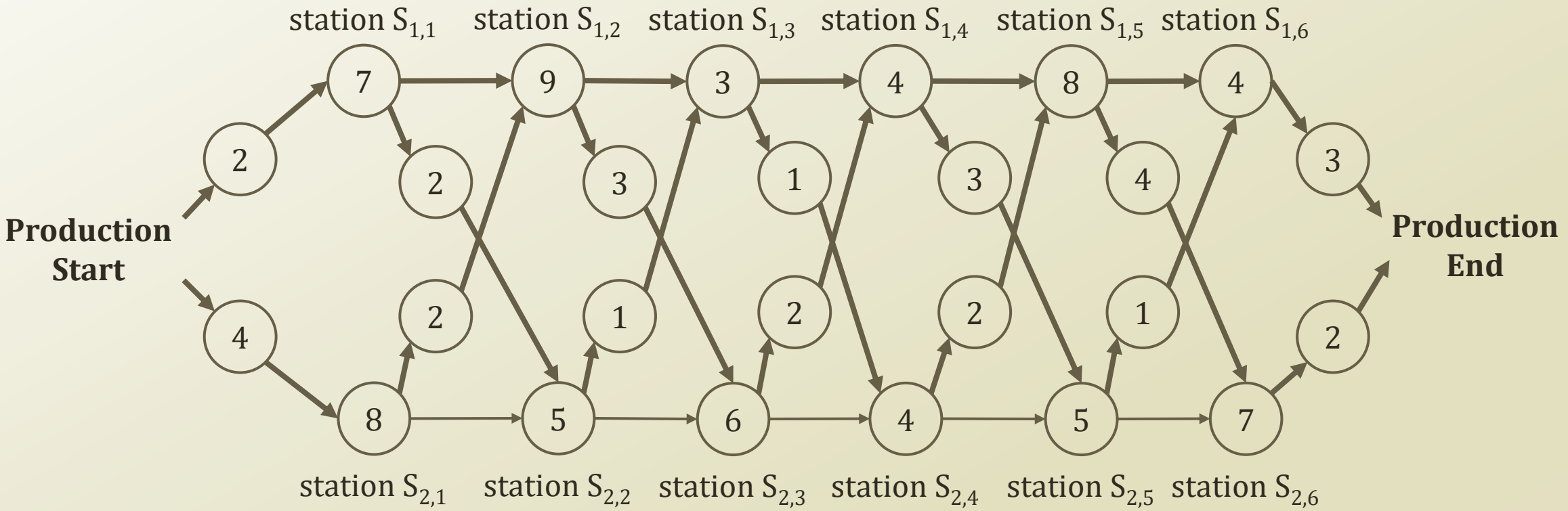
Process of Assembly Line Scheduling



Time	1	2	3	4	5	6
L1	9	18	20			
L2	12	16	22			

Trace	1	2	3	4	5	6
L1	S	1	2			
L2	S	1	2			

Process of Assembly Line Scheduling



Time	1	2	3	4	5	6
L1	9	18	20	24	32	35
L2	12	16	22	25	30	37

Trace	1	2	3	4	5	6
L1	S	1	2	1	1	2
L2	S	1	2	1	2	2

Implementation Example: Assembly Line Scheduling in Recursion

```
class AssemblyLines:
    timeStation = [[7,9,3,4,8,4], [8,5,6,4,5,7]]
    timeBelt = [[2,2,3,1,3,4,3],[4,2,1,2,2,1,2]]
    intCount = 0
    def Scheduling(self, idxLine, idxStation):
        print("Calculate scheduling : line, station : ", idxLine, idxStation, "(", self.intCount, "recursion calls )")
        self.intCount = self.intCount + 1
        if idxStation == 0:
            if idxLine == 1:
                return self.timeBelt[0][0] + self.timeStation[0][0]
            elif idxLine == 2:
                return self.timeBelt[1][0] + self.timeStation[1][0]
        if idxLine == 1:
            costLine1 = self.Scheduling(1, idxStation-1) + self.timeStation[0][idxStation]
            costLine2 = self.Scheduling(2, idxStation-1) + self.timeStation[0][idxStation] + self.timeBelt[1][idxStation]
        elif idxLine == 2:
            costLine1 = self.Scheduling(1, idxStation-1) + self.timeStation[1][idxStation] + self.timeBelt[0][idxStation]
            costLine2 = self.Scheduling(2, idxStation-1) + self.timeStation[1][idxStation]
        if costLine1 > costLine2:
            return costLine2
        else:
            return costLine1

    def startScheduling(self):
        numStation = len(self.timeStation[0])
        costLine1 = self.Scheduling(1, numStation - 1) + self.timeBelt[0][numStation]
        costLine2 = self.Scheduling(2, numStation - 1) + self.timeBelt[1][numStation]
        if costLine1 > costLine2:
            return costLine2
        else:
            return costLine1
```

```
lines = AssemblyLines()
time = lines.startScheduling()
print("Fastest production time :", time)
```

Setting up recursion calls

Implementation Example: Assembly Line Scheduling in DP

```python
class AssemblyLines:
    timeStation = [[7,9,3,4,8,4], [8,5,6,4,5,7]]
    timeBelt = [[2,2,3,1,3,4],[4,2,1,2,2,1,2]]

    timeScheduling = [list(range(6)),list(range(6))]
    stationTracing = [list(range(6)),list(range(6))]
    def startSchedulingDP(self):
        numStation = len(self.timeStation[0])
        self.timeScheduling[0][0] = self.timeStation[0][0] + self.timeBelt[0][0]
        self.timeScheduling[1][0] = self.timeStation[1][0] + self.timeBelt[1][0]

        for itr in range(1,numStation):
            if self.timeScheduling[0][itr-1] > self.timeScheduling[1][itr-1] + self.timeBelt[1][itr]:
                self.timeScheduling[0][itr] = self.timeStation[0][itr] + self.timeScheduling[1][itr-1] + self.timeBelt[1][itr]
                self.stationTracing[0][itr] = 1
            else:
                self.timeScheduling[0][itr] = self.timeStation[0][itr] + self.timeScheduling[0][itr-1]
                self.stationTracing[0][itr] = 0

            if self.timeScheduling[1][itr-1] > self.timeScheduling[0][itr-1] + self.timeBelt[0][itr]:
                self.timeScheduling[1][itr] = self.timeStation[1][itr] + self.timeScheduling[0][itr-1] + self.timeBelt[0][itr]
                self.stationTracing[1][itr] = 0
            else:
                self.timeScheduling[1][itr] = self.timeStation[1][itr] + self.timeScheduling[1][itr-1]
                self.stationTracing[1][itr] = 1

        costLine1 = self.timeScheduling[0][numStation-1] + self.timeBelt[0][numStation]
        costLine2 = self.timeScheduling[1][numStation-1] + self.timeBelt[1][numStation]
        if costLine1 > costLine2:
            return costLine2, 1
        else:
            return costLine1, 0

    def printTracing(self, lineTracing):
        numStation = len(self.timeStation[0])
        print("Line :",lineTracing,", Station :", numStation)
        for itr in range(numStation-1, 0, -1):
            lineTracing = self.stationTracing[lineTracing][itr]
            print("Line :",lineTracing,", Station :", itr)
```

Setting up a memoization table

Building up a bigger solutions

```python
lines = AssemblyLines()
time, lineTracing = lines.startSchedulingDP()
print("Fastest production time :", time)
lines.printTracing(lineTracing)
```

Further Readings

- Introduction to Algorithms by Cormen et al.
  - pp. 62-76, 323-356