

# 4장. Recursions and Dynamain Programming



# 목차

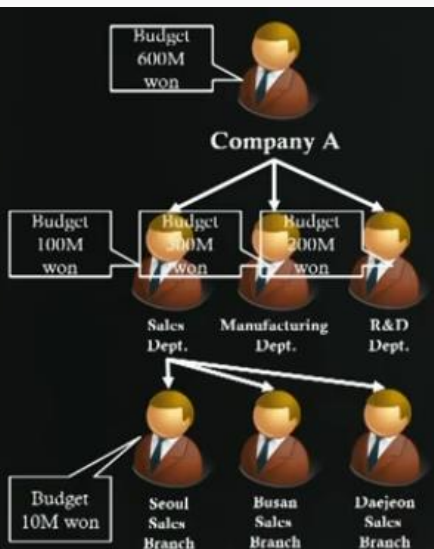
1. Recursions
2. Merge Sort and Problems in Recursions
3. Dynamic Programming
4. Fibonacci Sequence in DP
5. Process of Assembly Line Scheduling
6. Assembly Line Scheduling in Recursion and DP

# 1. Recursions

## 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



- 계층적으로 조직이 있어서 어떻게 쪼개지는지를 볼수 있다.
- 위에 뷰와 아래의 뷰는 유사하다. 달라진것은 예산의 크기가 줄어들어다
- **repeation problem**은 이렇게 큰 하나의 문제가 정의 되면 그 문제를 숙의해서 다시 반복되는 또 다른 문제로 만든다는 것이다. 다만 그 문제가 작아지게끔 만드는 것이다.
- 문제를 잘게 쪼개어서 문제를 해결해 나가는 과정을 **divide and conquer**라고 한다.
- 상위에도 **burget** 계산이 있고 밑에도 **burget** 계산이 있어서 고 상위 함수를 하위에서 동일한 함수를 콜해서 사용한다. 다만 둔화된 디바이드된 작은 문제로 동일한 함수를 콜하는 것이 **recursion**이 되고 그게 리피팅 프라블럼을 프로그램으로써 풀어보는 과정이 되고 그리고 그 컨셉은 디바이드 컨셉을 따르는 것이다.

# 1. Recursions

## More examples

### ◆ Factorial

$$\diamond \text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times (n-1) \times \dots \times 2 \times 1 & \text{if } n > 0 \end{cases}$$

### ◆ Repeating problems?

$$\diamond \text{Factorial}(n) = \begin{cases} 1 & \text{if } n = 0 \\ n \times \text{Factorial}(n-1) & \text{if } n > 0 \end{cases}$$

### ◆ Great Common Divisor

$$\diamond \text{GCD}(32, 24) = 8$$

### ◆ Euclid's algorithm

$$\diamond \text{GCD}(A, B) = \text{GCD}(B, A \bmod B)$$

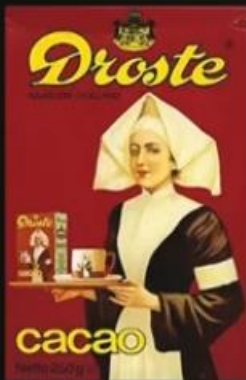
$$\diamond \text{GCD}(A, 0) = A$$

### ◆ Commonality

◆ Repeating function calls

◆ Reducing parameters

◆ Just like the mathematical induction



Self-Similar

- Factorial !
- 팩토리아 속에 평선이 n-1이 되면 또 같은 평선이 나온다.
- great common divisor 최대 공약수
- gcd(32, 24)에서 32를 24로 나누는 몫은 1이고 나머지는 8
- gcd(24, 8)에서 24를 8로 나누면 몫은 3이고 나머지는 0
- gcd(8, 0) -> 최대공약수는 8
- 사이즈는 작아지고 있고 평선은 동일하다
- 공통점은 평선꼴은 반복되고 사이즈는 줄어든다.

# 1. Recursions

## 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)

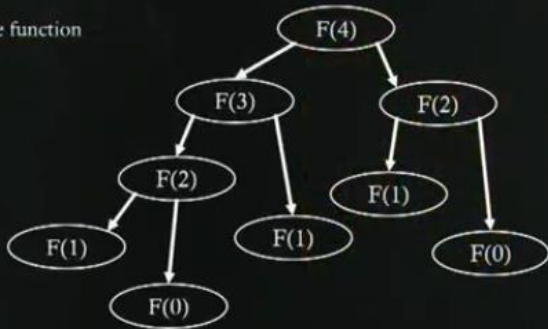
◆ Return A;

```
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),
```

0 1 1 2 3 5 8 13 21 34

Program Execution Flow



Fibonacci(n)

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

- recursion은 리피팅하는 같은 방법으로 핸들하는것이다.
- **sewdo code** 실제로 안돌아가는데 이렇게하는것이다 라고 간략히 적어놓것이다.
- **fuctionA**를 콜해서 사용하지만 **target**은 사이즈가 작아지는 것이다.

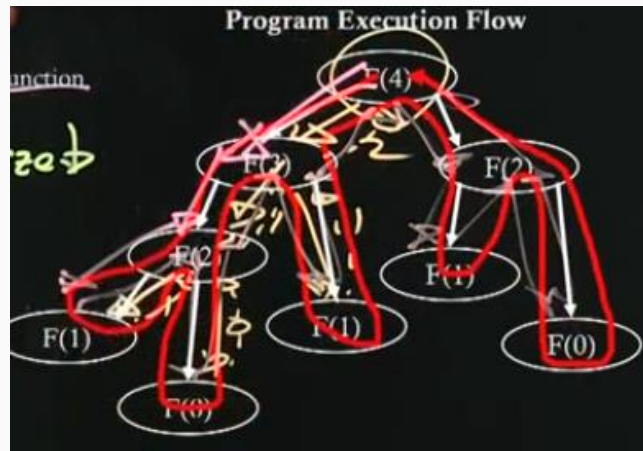
# Recursion

```
In[36]:
...: for itr in range(0, 10):
...:     print(Fibonacci(itr))
...:
```

```
def fib(10):
    for itr in range(0, 10):
        print(Fibonacci(itr))

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

- 0 1 1 2 5 8 13 21 34
- 앞에 0과 1은 규칙에 따라서 작성한다.
- 두개를 가지고 다음 값을 만든다.
- 정의한 함수를 두번 콜하고 있다.
- 사이즈는 계속 죽어든다.
- 탈출구문은 n이 0이거나 1이면 더이상 자기 자신을 재귀 리  
컬전을 하지 않고 탈출하게 된다.



# 1. Recursions

## Recursions and Stackframe

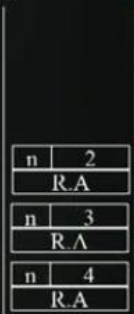
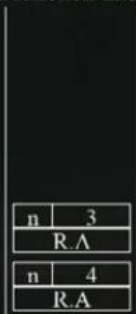
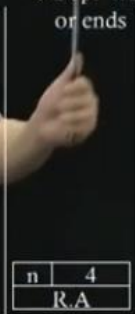
### ◆ Recursion of functions

#### ◆ Increase the items in the stackframe

◆ Stackframe is a stack storing your function call history

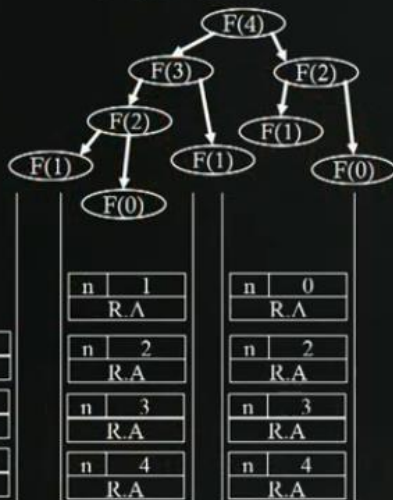
◆ Push: When a function is invoked

◆ Pop: When a function hits *return* or ends



### ◆ What to store?

◆ Local variables and function call parameters



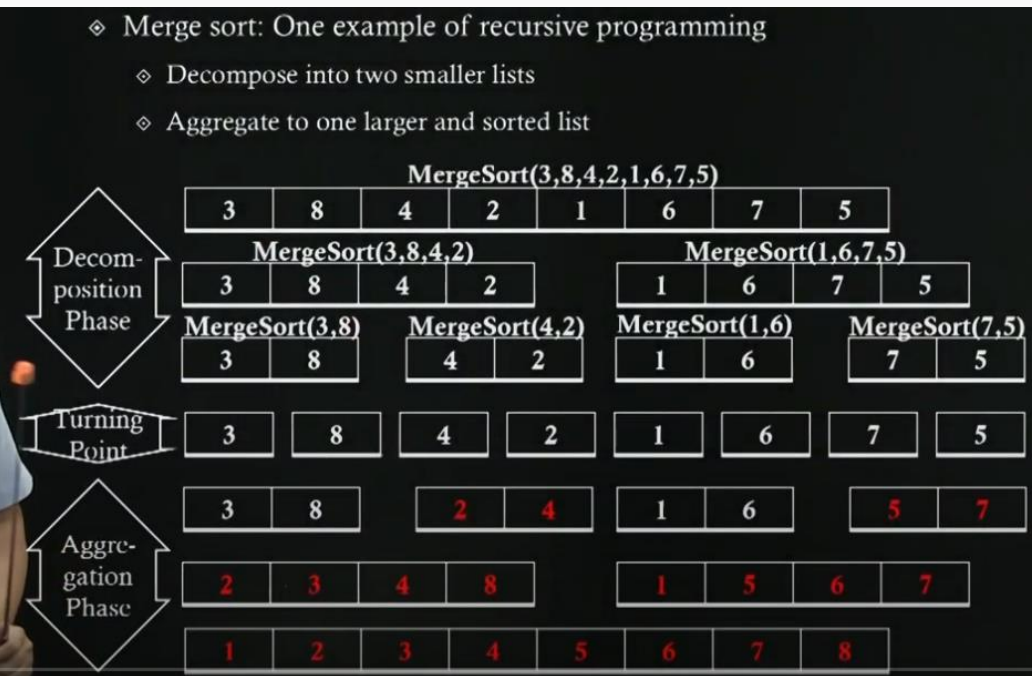
- recursion은 재귀호출이 계속 일어나는것
  - **stackframe**은 **stack**인데 특별한 목적을 가진 스택이다. 평션 콜의 히스토리의 진행된 역사를 기록하고 있는 것을 스택 프레임이라고 한다.
  - 스택에 대해서 할 수 있는 오퍼레이션은 푸쉬와 팝 두가지만 있다.
  - 푸쉬라는 것은 평션이라는 것이 콜되면(invoked)
  - **Pop**은 **function**이 리턴을 타 가지고 평션을 받고 나가게 되면 그러면 **Pop**이 일어난다.
  - 어떤것을 저장해 놓느냐가 또한 핵심인데 이것은 로컬 베리어 불과 평션 파라미터를 저장해놓고 해야 된다.
  - **Local Variables**은 평션 속에서만 접근 가능한 **within** 평션 버라이블이 된다.
  - 평션 콜 파라미터는 특정 평션 콜 인스턴스에 할당된 파라미터 예를 들어 **f4**라고 하면 이자가 바로 평션 콜의 파라미터가 된다.
- ? 진행방향이 왜 4-3-2-1-0으로 가는지 모르겠습니다.

# 2. Merge Sort and Problems in Recursions

## Merge Sort

◆ Merge sort: One example of recursive programming

- ◆ Decompose into two smaller lists
- ◆ Aggregate to one larger and sorted list



- 리컬전을 이용한 **sort algorithms** 중의 하나입니다.
- 기본적인 원리는 이것을 더 작은 리스트 사이즈로 줄이고 줄이고 해서 더 이상 쪼갤 수 없는 한 숫자가 되었을 때 비교해가면서 리스트를 합쳐 나가는 과정 이 **merge sort** 과정입니다.
- 수열이 **38421675**
- 중간을 쪼갬다 2와 1사이
- 누군가 이 리스트를 가지고 **merge sort**를 콜하면 **merge sort** 내부에서 중간을 자르고 나눠진 부분을 또 콜한다.
- 다음 다시 리컬전으로 반반씩 자른다.
- 더 이상 쪼갤수 없을때 **aggregation**하는 합쳐나가는 과정을 한다.
- 두개의 리스트를 비교해 나간다.
- 작은거 먼저 내려오고 다음 8이 내려온다
- 앞에 있는것끼리 비교 2와 3비교, 3과 4비교, 8과 4비교



# 2. Merge Sort and Problems in Recursions

## Implementation Example: Merge Sort

```
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] = lstSubElementToSort1[idxCount1]
            idxCount1 = idxCount1 + 1
        else:
            lstElementToSort[itr] = lstSubElementToSort2[idxCount2]
            idxCount2 = idxCount2 + 1
    return lstElementToSort
```

**Execution Code**

```
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

- performMergeSort function이 밑에서 두개의 평선으로 일어난다.
- 두개로 쪼개지기 때문에 평선이 두개로 나뉜다.
- 사이즈는 줄어든다.
- 위에서 리스트를 만든다.
- 절반 사이즈 전에는 첫번째 리스트, 넘어가면 두번째 리스트
- 중간 부분 이후에는 aggregation하는 부분

# 2. Merge Sort and Problems in Recursions

## Problems in Recursions of Fibonacci Sequence

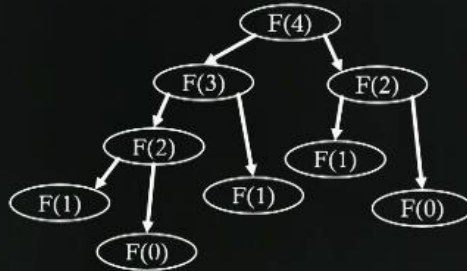
### ◆ Problems in recursions

#### ◆ 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?



- 평선 콜이 너무 많다.
- 불필요하게 너무나 많은 시간과 메모리 공간을 잡아먹게 된다.
- 그래서 해결책으로 **Dynamic Programming**을 사용한다.
- 이것은 한번 콜하고 콜한 결과를 기록하는 것이다.

# 3. 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)	?

- 다이나믹 프로그래밍을 통해서 불필요한 평선콜을 없앨 수 있는지 알아보겠다.
- Dynamic programming은 overlapping sub-instances가 있는 recurrence를 푸는 방식이다
- 다이나믹 스케줄링, 플랜닝이라고도 다른 분야에선 부른다.
- f4하기 위해서 f2나 f1이 오버랩핑 되고 있다.
- 스몰 인스턴스를 하나 먼저 풁니다.
- 큰 문제를 푸는데 문제와 관련된 스몰 인스턴스를 아주 작은 레벨에서 부터 풀어 나가는 것이다.
- f4를 먼저 풀기 전에 f1이나 f0를 먼저 푸는 것이다.
- 그 다음 결과를 저장한다.

# 3. Dynamic Programming

## Memoization

### ◆ Key technique of dynamic programming

#### ◆ Simply put

- ◆ Storing the results of previous function calls to reuse the results again in the future

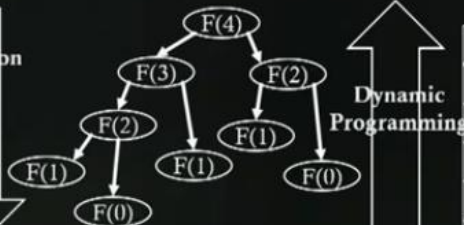
#### ◆ More philosophical sense

- ◆ Bottom-up approach for problem-solving
  - ◆ Recursion: Top-down of divide and conquer
  - ◆ Dynamic programming: Bottom-up of storing and building

Stackframe

n	2
RA	
n	3
RA	
n	4
RA	

Recursion



Dynamic Programming

Memoization

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

- 다이나믹 프로그래밍은 결과를 기록하여 사용하는 것이고 기존의 평선콜과 그것을 **result**를 재사용하기 위해서 결과를 저장하는 것이다.
- 그래서 테이블을 만드는 데 그것이 **Memoization table**이 된다.
- **Memoization**과 **recursion**이 돌아가기 위해서 만든 **stack** 프레임이 있는데 두개의 상반된 철학적인 것이 있다.
- 하나는 **bottom-up approach**
- **memoization**이 **bottom-up approach**다 밑에서부터 올라가는 것
- **top-down approach**는 **recursion** 이다.
- **f4**를 풀려고 먼저 시도하고 그것을 풀기 위해 **f3 f2**를 콜해서 푸는 방식
- **stack**을 이용한 **recursion** 방식과 **memoization**을 이용한 **dynamic** 방식은 상반된다.

# Implementation Example: Fibonacci Sequence in DP

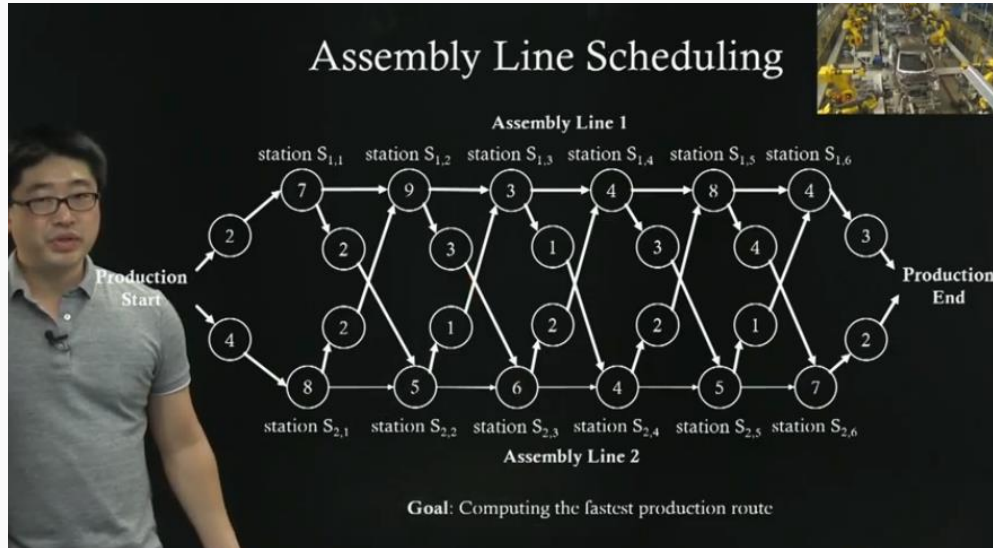
- DP를 활용한 Fibonacci 코드입니다.
- 규칙은 0과 1을 그대로 적어둡니다.
- Memoization은 key와 value의 dict 형태로 저장됩니다.

- ◆ 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  $O(N)$

Windows 정품 인증  
[설정]으로 이동하여 Windows를 정품 인증합니다.

# 5. Process of Assembly Line Scheduling

## Assembly Line Scheduling



- 시간이 맞지 않아서 새로운 제품이 들어갈때 어느 줄로 가야 가장 빠리 나올것인지 이 문제를 해결하는 데 **dynamic programming**을 쓰게 된다.
- 라인은 두개로 되어있고 한쪽 라인이 시간이 걸릴경우 다른 라인으로 가는 길이 설계되어 있다.
- 동그라이 안에 있는 숫자는 공정이 걸리는 시간이다.
- 중간 시간은 옆 라인으로 가는 시간

# 5. Process of Assembly Line Scheduling

## 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
```

- Function은 Scheduling
- 탈출코드 – if idxStation 부분
- Recursive call = Function call 부분 – self.scheduling



# 5. Process of Assembly Line Scheduling

## Implementation Example: Assembly Line Scheduling in DP

```
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]]

    timeScheduling = [range(6), range(6)]
    stationTracing = [range(6), 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

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

- timeScheduling – memorization table def
- stationTracing - memorization table def
- Self.timeScheduling - memorization
- 증가하는 과정은 for loop
- numStation – initialization(초기화 하는부분)