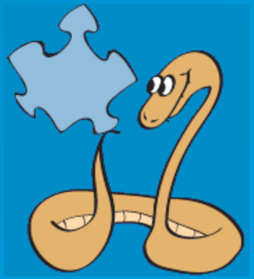
파이썬 알고리즘

ALGORITHMS IN PYTHON

최영규 지음



CHAPTER

07

동적 계획법

학습 내용

7.1 피보나치수열과 동적 계획법

7.2 이항계수 구하기

7.3 배낭 채우기 문제: 0-1 Knapsack

7.4 최장 공통 부분순서 문제

7.5 그래프의 인접 행렬 표현과 최단 경로 문제

7.6 모든 정점간의 최단 경로 길이

7.7 편집 거리

동적 계획법

- Dynamic programming
  - 1950년대에 미국의 수학자인 벨맨(Richard Bellman)
  - 다단계의 의사 결정 프로세스를 최적화하는 일반적인 방법
  - → 동적 계획법으로 번역

- 동적 계획법
  - 정복 기법과 유사한 부분이 많음
  - 부분 문제들의 답을 저장, 다시 꺼내서 사용

  → 같은 부분 문제를 다시 풀지 않도록 하는 것이 핵심
  - 공간으로 시간을 버는 전략의 일종

7.1 피보나치수열과 동적 계획법

- 피보나치 수열 (분할 정복)

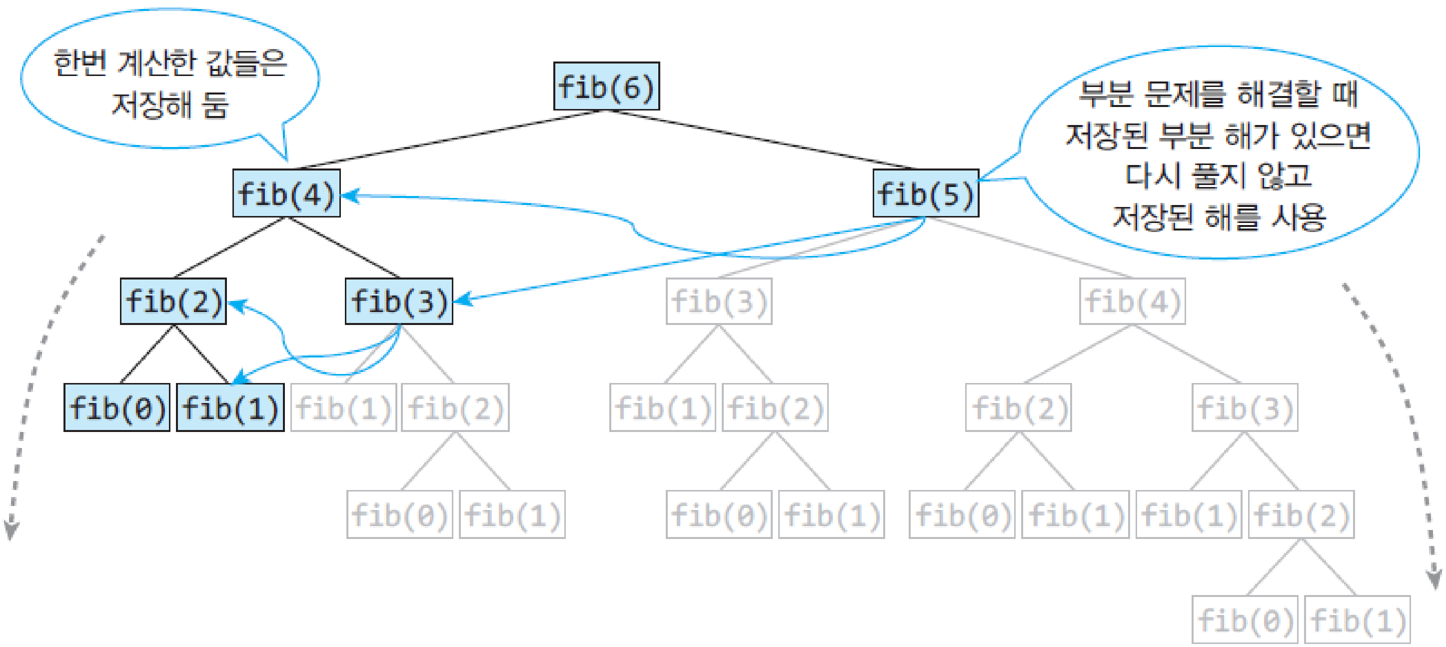
$$fib(n) = \begin{cases} n & n \le 1 \\ fib(n-2) + fib(n-1) & \text{otherwise} \end{cases}$$

  - 심각한 계산의 중복: → $O(2^n)$

- 동적 계획법
  - 어렵게 구한 답을 한 번만 쓰고 버리지 말고 저장→ 재사용
  - 구현 방법
    - 메모이제이션(memoization): 하향식(top-down)
    - 테이블화(tabulation): 상향식(bottom-up)

메모이제이션을 이용한 피보나치수열

문제를 풀 때마다 이미 풀린 문제인지를 먼저 확인한다. 풀린 문제이면 저장된 답을 이용하고, 풀리지 않은 문제이면 그 문제를 풀고 답을 다시 저장한다.



알고리즘과 복잡도

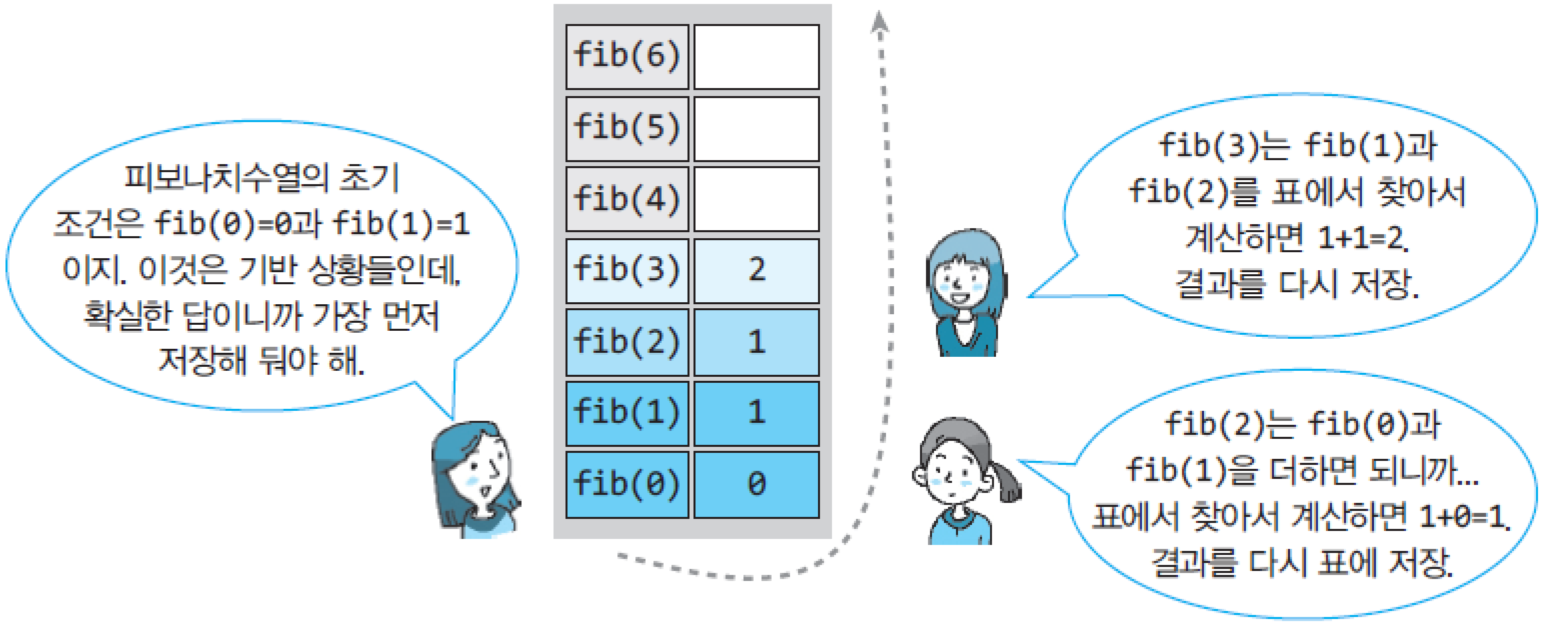
알고리즘 7.1 피보나치수열(메모이제이션 이용)

```
def fib_dp_mem(n) :
    if( mem[n] == None ) :              # 풀리지 않은 경우 → 계산하고 저장
        if n < 2 :
                mem[n] = n              # 기반 상황: n<=1
        else:                           # 일반 상황: otherwise
                mem[n] = fib_dp_mem(n-1) + fib_dp_mem(n-2)
    return mem[n]
```

- 시간 복잡도
  - $O(n)$
- 공간 복잡도
  - $O(n)$

테이블화를 이용한 피보나치수열

결과를 저장할 테이블을 먼저 만든다. 다음으로 답이 이미 알려진 단순한 상황, 즉 **기반 상황** (base case)에 대한 테이블 항목을 먼저 채우고, 이들을 바탕으로 테이블을 채워서 올라간다.



알고리즘과 복잡도

알고리즘 7.2 피보나치수열(테이블화

```
def fib_dp_tab(n) :
    f = [None] * (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]
```

- 시간 복잡도
  - $O(n)$
- 공간 복잡도
  - $O(n)$

알고리즘 테스트 테이블화와 메모이제이션을 이용한 피보나치수열

```
n = 8
print('동적계획(  테이블화  ): Fibonacci(%d) = '%n, fib_dp_tab(n))
mem = [None] * (n+1)
print('동적계획(메모이제이션): Fibonacci(%d) = '%n, fib_dp_mem(n))
```

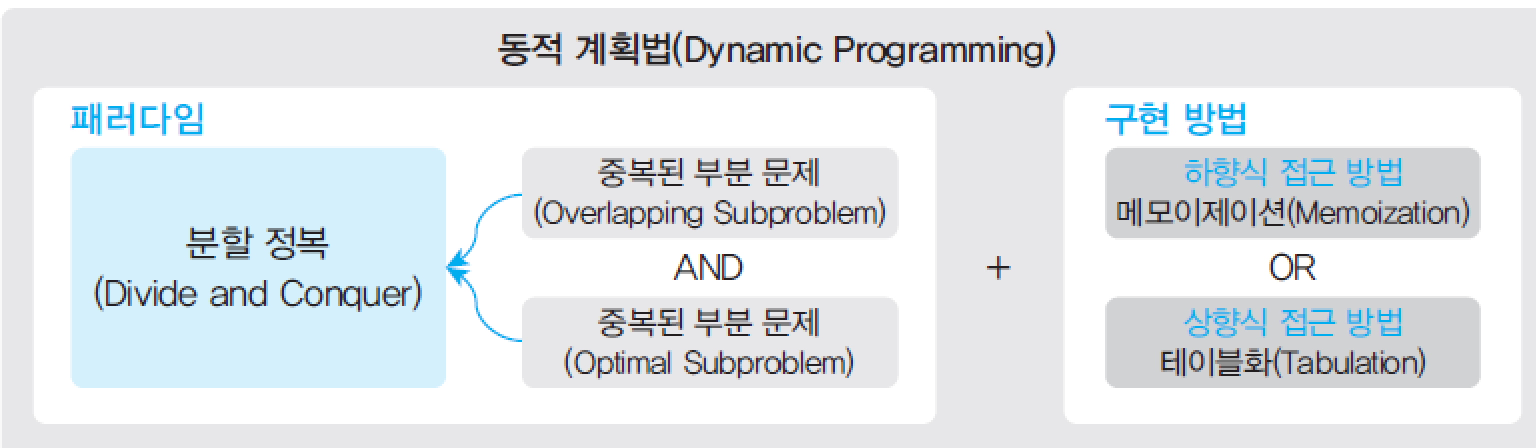
C:\WINDOWS\system32\cmd.exe

```
동적계획(  테이블화  ): Fibonacci(8) =  21
동적계획(메모이제이션): Fibonacci(8) =  21
```

동적 계획법 패러다임

- 문제의 특성
  - (1) 겹치는 부분 문제(overlapping subproblem)
  - (2) 최적 부분 구조(optimal substructure)
    - 부분 문제의 최적해들을 이용하면 전체문제의 최적해를 구할 수 있는 구조

- 패러다임과 구현 방법



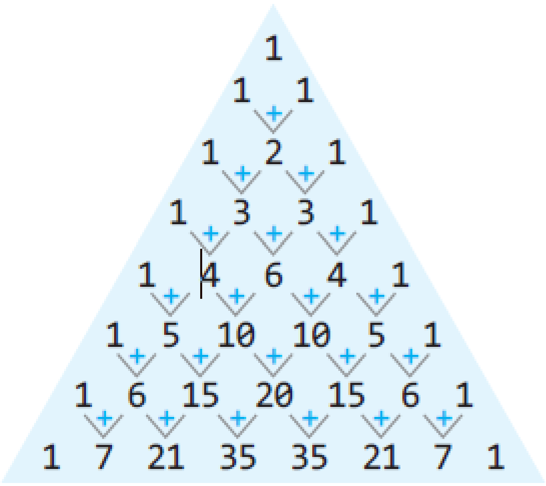
7.2 이항계수 구하기

- 이항정리와 파스칼의 삼각형
  - 이항 정리

$$(a+b)^n = \sum_{k=0}^{n} C(n,\, k)\, a^{n-k} b^k$$

  - 이항 계수

$$C(n,\, k) \text{는 } \frac{n!}{k!(n-k)!}$$



➔ 동적 계획법에 의한 풀이

이항계수의 순환 관계식

- $C(n,k)$: n 개에서 k 개를 뽑는 순서 없는 조합의 가짓수

- 순환 관계식 만들기
  - 기반 상황
    - 답이 이미 알려진 가장 작은 부분 문제
    - $C(n,0)$ = 1
    - $C(n,n)$ = 1
  - 일반 상황
    - n 번째 항목을 포함하는 경우: $C(n-1,k-1)$
    - n 번째 항목을 포함하지 않는 경우: $C(n-1,k)$
  - 순환 관계식

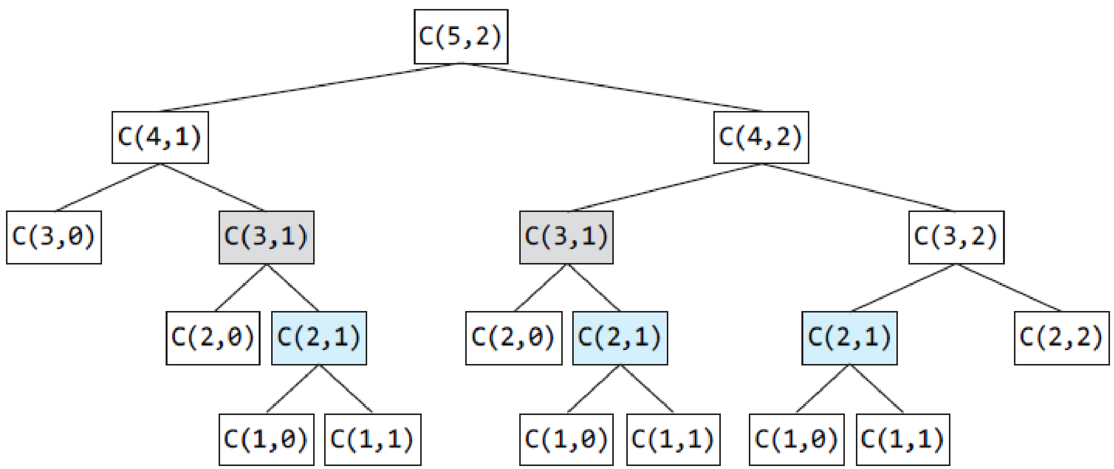
$$C(n,k)=\begin{cases}1 & k=0,\ \ k=n\\ C(n-1,\ k-1)+C(n-1,\ k) & \text{otherwise}\end{cases}$$

분할 정복에 의한 이항계수

알고리즘 7.3 이항계수 C(n, k) 계산 함수(분할 정복 기법)

```
def bino_coef_dc(n , k):
    if k==0 or k ==n :
        return 1
    return bino_coef_dc(n-1 , k-1) + bino_coef_dc(n-1 , k)
```

- 많은 중복이 발생



동적 계획법에 의한 이항계수

- 테이블
  - 두 개의 변수 → 2차원 테이블
  - 예: $C(5,3)$

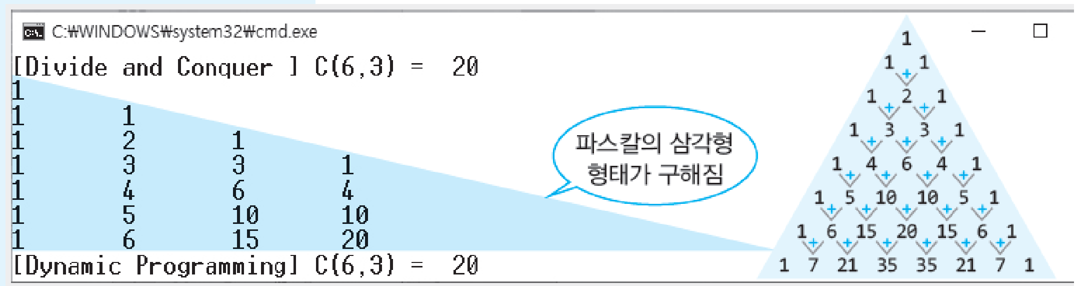
	$k=0$	$k=1$	$k=2$	$k=3$
$n=0$	1			
$n=1$	1	1		
$n=2$	1	2	1	
$n=3$	1	3	3	1
$n=4$	1	4	6	4
$n=5$	1	5	10	$\underline{10}$

알고리즘

알고리즘 7.4 이항계수 C(n, k) 계산 함수(동적 계획법)

```
def bino_coef_dp(n, k):
    C = [[-1 for _ in range(k+1)] for _ in range(n+1)]

    for i in range(n+1):
        for j in range(min(i, k)+1):
            if j == 0 or j == i:
                C[i][j] = 1
            else:
                C[i][j] = C[i-1][j-1] + C[i-1][j]
    return C[n][k]
```



- 시간 복잡도: $O(nk)$
- 공간 복잡도: $O(nk)$

7.3 배낭 채우기 문제: 0-1 Knapsack

- 억지기법: 완전 탐색(문제 3.7)
  - 원소의 개수가 n인 집합의 부분집합의 수가 2^n → $O(2^n)$

- 배낭 문제의 순환 관계식 만들기
  - 배낭 용량 W
  - n개의 물건: E_1, E_2, …, E_n
  - $E_i = (wt_i, val_i)$
  - $A(n, W)$: 배낭의 최대 가치

배낭 문제의 순환 관계식 만들기

- $A(k, w)$의 기반 상황
  - $A(0, w) = 0$, $A(k, 0) = 0$
- $A(k, w)$의 일반 상황
  - Case1: $wt_k > w$ → 넣을 수 없음
    - $A(k, w) = A(k-1, w)$
  - Case2: $wt_k \leq w$ → 넣거나, 안 넣거나
    - 넣은 경우: $A(k-1, w - wt_k) + val_k$
    - 넣지 않은 경우: $A(k-1, w)$

- 순환 관계식

$$\boxed{A(n, W) = \begin{cases} 0 & \text{if } W = 0 \text{ or } n = 0 \\ A(n-1, W) & \text{if } wt_n > W \\ \max(val_n + A(n-1, W - wt_n), A(n-1, W)) & \text{otherwise} \end{cases}}$$

분할 정복 알고리즘

알고리즘 7.5 0-1 배낭 채우기 알고리즘(분할 정복 기법)

```
def knapSack_bf(W, wt, val, n):
    if n == 0 or W == 0 :       # 기반 상황
        return 0

    if (wt[n-1] > W):
        return knapSack_bf(W, wt, val, n-1)
    else:
        valWithout = knapSack_bf(W, wt, val, n-1)
        valWith = val[n-1] + knapSack_bf(W-wt[n-1], wt, val, n-1)
        return max(valWith, valWithout)
```

- 많은 중복 발생

동적 계획법에 의한 0-1 배낭 문제 해법

- 문제에 대한 추가적인 제한
  - 무게의 단위를 정수로 제한 → 동적 계획법 적용 가능
- 테이블 설계
  - $(n+1)*(W+1)$

		배낭 용량						
		0	1	…	$w-wt_i$	w	…	W
물건의 수	0	0	0	…	0	0	…	0
	1	0						
	…	…						
	$i-1$	0			$A(i-1, w-wt_i)$	$A(i-1, w)$		
	i	0				$A(i, w)$		
	…	…						
	n	0						$A(n, W)$

알고리즘

알고리즘 7.6 0-1 배낭 채우기 알고리즘(동적 계획법)

```
def knapSack_dp(W, wt, val, n):
    A = [[0 for x in range(W + 1)] for x in range(n + 1)]

    for i in range(1, n + 1):
        for w in range(1, W + 1):
            if wt[i-1] > w:
                A[i][w] = A[i-1][w]
            else :
                valWith = val[i-1] + A[i-1][w-wt[i-1]]
                valWithout = A[i-1][w]
                A[i][w] = max(valWith, valWithout)

    return A[n][W]
```

알고리즘 테스트 0-1 배낭 채우기 문제에 대한 분할 정복과 동적 계획법

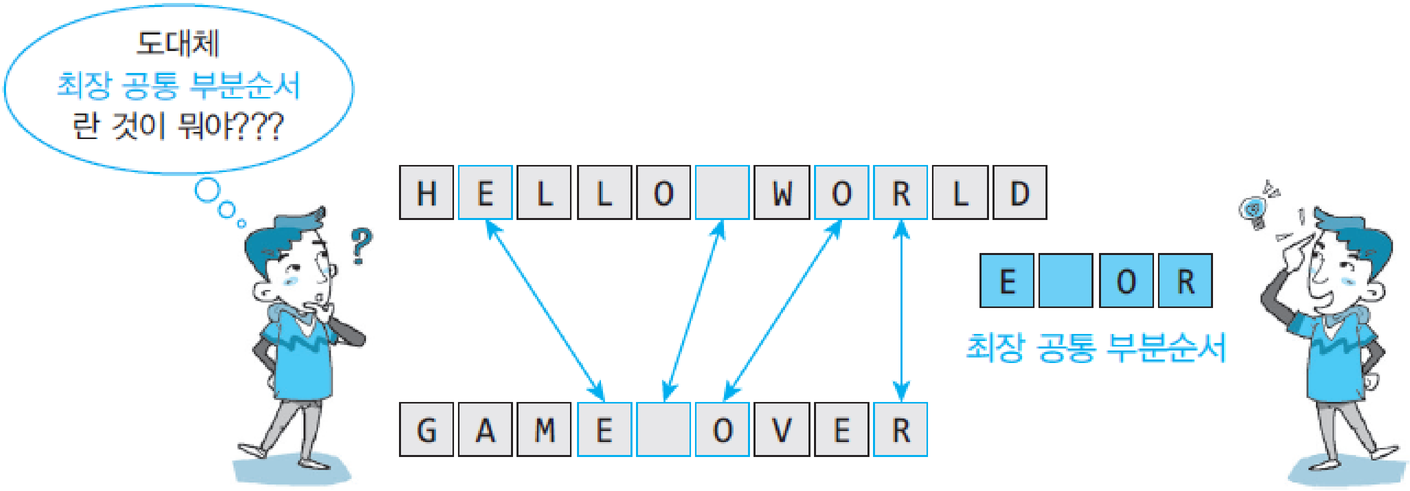
```
val = [60, 100, 190, 120, 200, 150]
wt = [2, 5, 8, 4, 7, 6]
W = 18
n = len(val)
print("0-1배낭문제(분할 정복): ", knapSack_bf(W, wt, val, n))
print("0-1배낭문제(동적 계획): ", knapSack_dp(W, wt, val, n))
```

```
C:\WINDOWS\system32\cmd.exe
0-1배낭문제(분할 정복):  480
0-1배낭문제(동적 계획):  480
```

- 시간 복잡도: $O(nW)$
- 공간 복잡도: $O(nW)$

7.4 최장 공통 부분 순서 문제

- Longest Common Subsequence, LCS
  - 데이터의 유사도를 평가
  - 유전자 염기 서열, 두 소스 파일의 차이 등

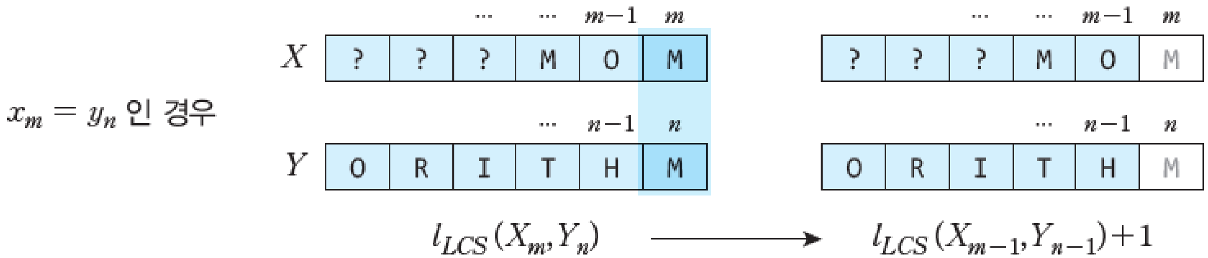


[그림 7.10] 최장 공통 부분순서 문제의 예

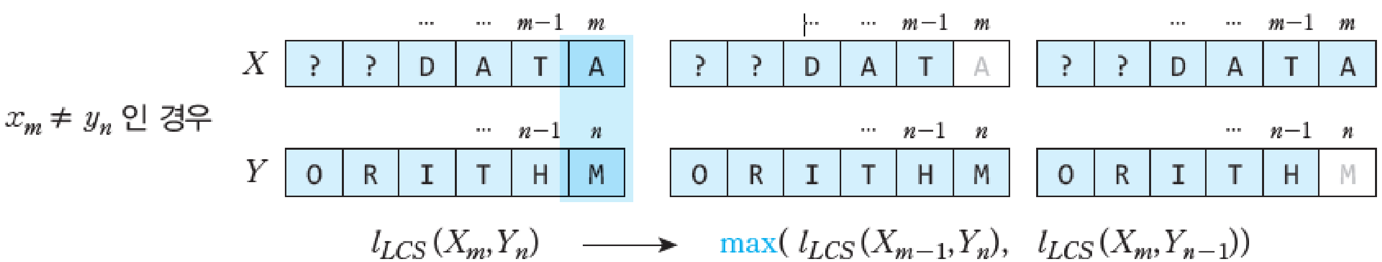
- LCS의 길이: $l_{lcs}(X_m, Y_n)$
  - $X =< x_1, x_2, \ldots, x_m >$, $Y =< y_1, y_2, \ldots, y_n >$

LCS 길이의 순환 관계식

- $l_{lcs}(X_m, Y_n)$ 의 기반 상황
  - $l_{lcs}(X_0, Y_n) = l_{lcs}(X_m, Y_0) = 0$
- $l_{lcs}(X_m, Y_n)$ 의 일반 상황
  - 마지막 문자를 중심으로 두 경우로 나눔



[그림 7.11] 마지막 문자가 같은 경우



[그림 7.12] 마지막 문자가 서로 다른 경우

순환구조 알고리즘

- 순환 관계식

$$l_{LCS}(X_m,\ Y_n) = \begin{cases} 0 & m=0 \ \text{ or } \ n=0 \\ 1 + l_{LCS}(X_{m-1},\ Y_{n-1}) & x_m = y_n \\ \max\left(l_{LCS}(X_{m-1},\ Y_n),\ l_{LCS}(X_m,\ Y_{n-1})\right) & \text{otherwise} \end{cases}$$

- 순환구조 알고리즘

알고리즘 7.7 최장 공통 부분순서(순환 구조)

```
def lcs_recur(X, Y, m, n):
    if m == 0 or n == 0:                    # base case
        return 0
    elif X[m-1] == Y[n-1]:                  # case 1: x_m == y_n
        return 1 + lcs_recur(X, Y, m-1, n-1)
    else:                                   # case 2
        return max(lcs_recur(X, Y, m, n-1), lcs_recur(X, Y, m-1, n))
```

동적 계획법에 의한 LCS 길이

- 테이블
  - $(m+1)*(n+1)$
  - $L[0][n] = L[m][0] = 0$

$$L[m][n] = \begin{cases} 0 & m = 0 \text{ or } n = 0 \\ 1 + L[m-1][n-1] & x_m = y_n \\ \max(L[m-1][n],\ L[m][n-1]) & \text{otherwise} \end{cases}$$

		H	E	L	L	O	' '	W	O	R	L	D
	0	0	0	0	0	0	0	0	0	0	0	0
G	0	0	0	0	0	0	0	0	0	0	0	0
A	0	0	0	0	0	0	0	0	0	0	0	0
M	0	0	0	0	0	0	0	0	0	0	0	0
E	0	0	1	1	1	1	1	1	1	1	1	1
' '	0	0	1	1	1	1	2	2	2	2	2	2
O	0	0	1	1	1	2	2	2	3	3	3	3
V	0	0	1	1	1	2	2	2	3	3	3	3
E	0	0	1	1	1	2	2	2	3	3	3	3
R	0	0	1	1	1	2	2	2	3	4	4	4

- 시간 복잡도 = 공간 복잡도: $O(mn)$

LCS 추적 알고리즘

- LCS 테이블에서 LCS 문자열을 추적하는 과정

		H	E	L	L	O	' '	W	O	R	L	D
	0	0	0	0	0	0	0	0	0	0	0	0
G	0	↑0	0	0	0	0	0	0	0	0	0	0
A	0	↑0	0	0	0	0	0	0	0	0	0	0
M	0	↑0	0	0	0	0	0	0	0	0	0	0
E	0	0	↖1	←1	←1	←1	1	1	1	1	1	1
' '	0	0	1	1	1	1	↖2	←2	2	2	2	2
O	0	0	1	1	1	2	2	2	↖3	3	3	3
V	0	0	1	1	1	2	2	2	↑3	3	3	3
E	0	0	1	1	1	2	2	2	↑3	3	3	3
R	0	0	1	1	1	2	2	2	3	↖4	←4	←4

알고리즘 테스트

알고리즘 테스트 LCS 길이 알고리즘 테스트

```
X = "GAME OVER"
Y = "HELLO WORLD"
print("X = ", X)
print("Y = ", Y)
print("LCS(분할 정복)", lcs_dc(X , Y, len(X), len(Y)))
print("LCS(동적 계획)", lcs_dp(X , Y) )
```

C:\WINDOWS\system32\cmd.exe

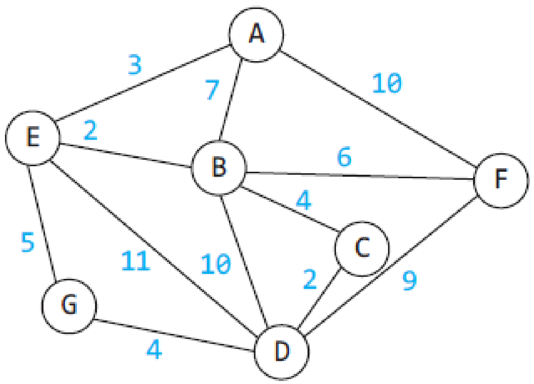
```
X =  GAME OVER
Y =  HELLO WORLD
LCS(분할 정복) 4
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1]
[0, 0, 1, 1, 1, 1, 2, 2, 2, 2, 2, 2]
[0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 3]
[0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 3]
[0, 0, 1, 1, 1, 2, 2, 2, 3, 3, 3, 3]
[0, 0, 1, 1, 1, 2, 2, 2, 3, 4, 4, 4]
LCS =  E OR
LCS(동적 계획) 4
```

7.5 그래프 인접 행렬 표현과 최단 경로 문제

- 최단 경로(shortest path) 문제
  - 가중치 그래프에서 두 정점을 연결하는 여러 경로들 중 에서 간선들의 가중치 합이 최소가 되는 경로를 찾는 문제
  - 예: "최단시간 경로", "최단거리 경로" 등

인접 행렬을 이용한 가중치 그래프의 표현

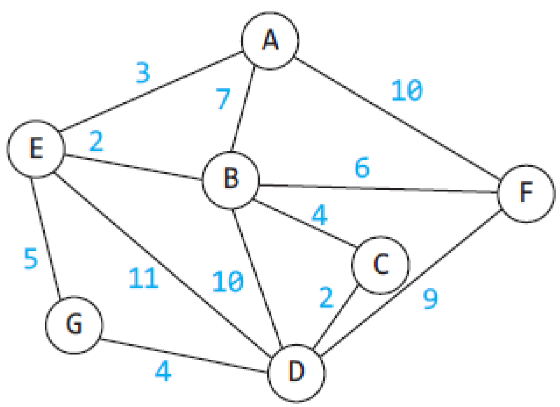
- MST 문제 등



가중치 그래프

```
vertex =  [ 'A',   'B',   C',  'D',  'E',  'F',  'G' ]
weight = [ [ None,    7,None,None,    3,   10,None ],
           [    7,None,    4,   10,    2,    6,None ],
           [ None,    4,None,    2,None,None,None ],
           [ None,   10,    2,None,   11,    9,    4 ],
           [    3,    2,None,   11,None,   13,    5 ],
           [   10,    6,None,    9,   13,None,None ],
           [ None,None,None,    4,    5,None,None ]]
```

- 최단 경로 문제

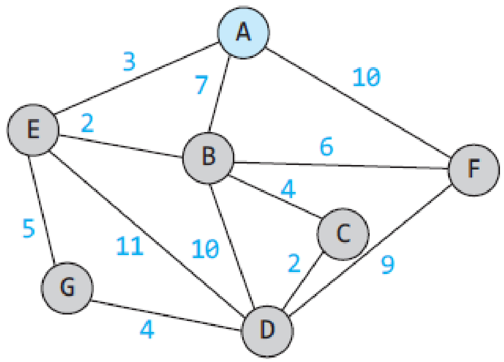


가중치 그래프

```
vertex =  [  'A',  'B',  'C',  'D',  'E',  'F',  'G' ]
weight = [ [    0,    7,  INF,  INF,    3,   10,  INF ],
           [    7,    0,    4,   10,    2,    6,  INF ],
           [  INF,    4,    0,    2,  INF,  INF,  INF ],
           [  INF,   10,    2,    0,   11,    9,    4 ],
           [    3,    2,  INF,   11,    0,   13,    5 ],
           [   10,    6,  INF,    9,   13,    0,  INF ],
           [  INF,  INF,  INF,    4,    5,  INF,    0 ]]
```

최단거리 문제의 종류

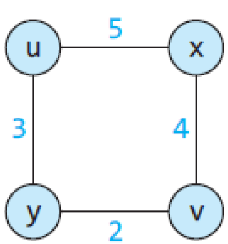
- 문제1) 시작 정점에서 도착 정점까지의 최단 경로 길이
- 문제2) 시작 정점에서 모든 정점까지의 최단 경로 길이




```
(A→A) 최단 경로 A = 0
(A→B) 최단 경로 A-E-B = 5
(A→C) 최단 경로 A-E-B-C = 9
(A→D) 최단 경로 A-E-B-C-D = 11
(A→E) 최단 경로 A-E = 3
(A→F) 최단 경로 A-F = 10
(A→G) 최단 경로 A-E-G = 8

결과: [ 0, 5, 9, 11, 3, 10, 8 ]
```

- 문제3) 모든 정점 간의 최단 경로 길이



가중치 그래프

```
V = [   'u', 'v', 'x', 'y' ]
W = [ [   0, INF,   5,   3],    # u
      [ INF,   0,   4,   2],    # v
      [   5,   4,   0, INF],    # x
      [   3,   2, INF,   0] ]   # y
```

인접 행렬 표현

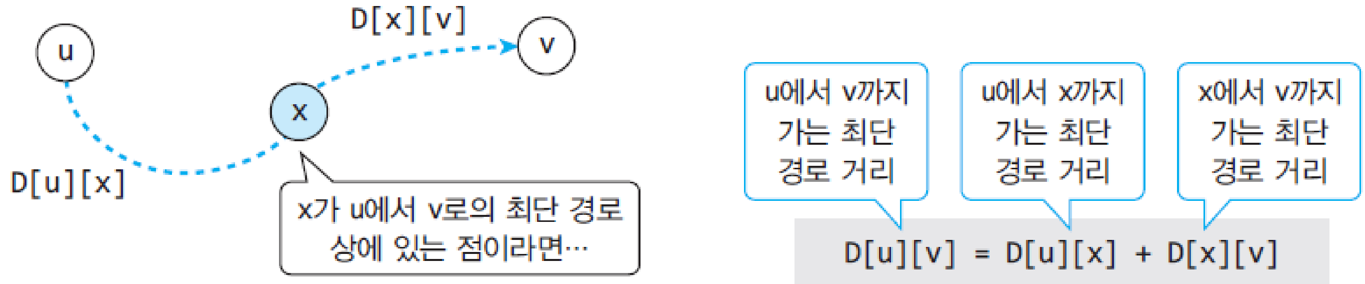
```
D = [ [ 0, 5, 5, 3],
      [ 5, 0, 4, 2],
      [ 5, 4, 0, 6],
      [ 3, 2, 6, 0] ]
```

계산 결과

[그림 7.19] 모든 정점에서 모든 정점까지의 최단경로 거리를 구하는 문제와 결과 예

7.6 모든 정점간의 최단 경로 길이

- Floyd-Warshall 알고리즘(Floyd 알고리즘)
  - 동적 계획법 사용
- 최단 경로 문제: 최적 부분 구조 특성을 만족함



[그림 7.20] 최단경로 문제는 최적 부분 구조 특성을 만족함

  - 최장 경로 문제
    - 최적 부분 구조 특성을 만족하지 않음

순환 관계식

- 인접 행렬 표현 그래프 *W* → D
  - 최단거리 행렬 *D*
  - $D^k[i][j]$:
    - 1~*k*번째 정점까지 만을 이용한 정점 *i* → *j*의 최단 경로
  - 최적해: D^n
- 동적 계획 전략
  - $D^0 \rightarrow D^1 \rightarrow D^2 \rightarrow \ldots \rightarrow D^n$

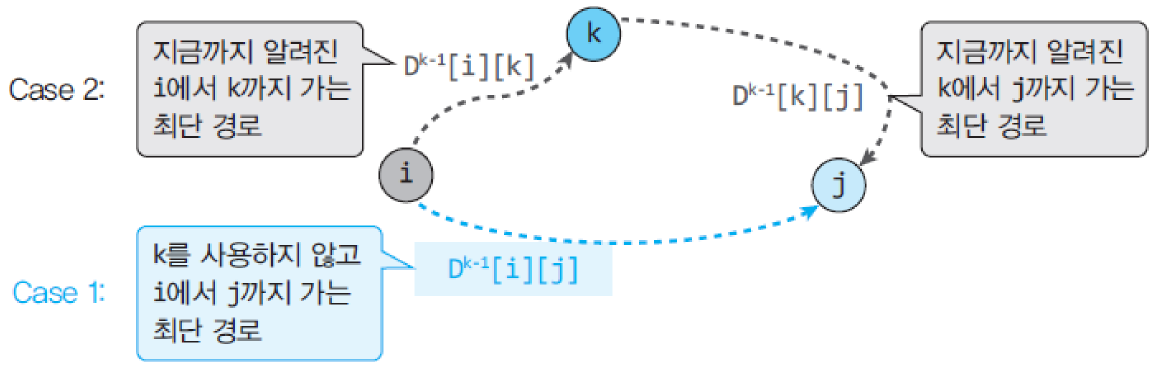
순환 관계식

- 기반 상황
  - D^0 : 아무런 정점을 거치지 않는 경로 → W
- 일반 상황
  - D^k : k까지의 정점만을 이용한 부분 문제의 최적해
  - D^{k-1}이 구해진 상태에서 k 번째 정점을 고려할때
    - 후보1: k 를 거치지 않는 경로

$$D^k[i][j] \leftarrow D^{k-1}[i][j]$$

    - 후보2: k 를 거치는 경로

$$D^k[i][j] \leftarrow D^{k-1}[i][k] + D^{k-1}[k][j]$$



알고리즘

- 순환 관계식

$$D^k[i][j] = \begin{cases} W[i][j] & k = 0 \\ \min\left(D^{k-1}[i][j],\ D^{k-1}[i][k] + D^{k-1}[k][j]\right) & \text{otherwise} \end{cases}$$

case1지금까지 알고 있는 경로

Case2 k를거쳐 들어오는 경로

- 알고리즘
  - 3중 루프

알고리즘 7.10 Floyd의 최단경로 탐색 알고리즘

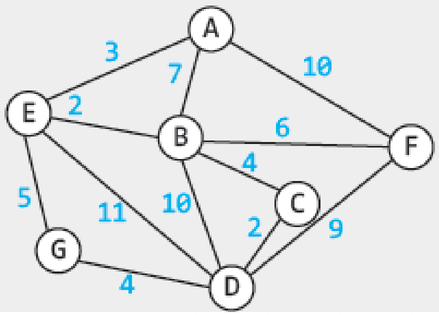
```
import copy
def shortest_path_floyd(vertex, W):
    vsize = len(vertex)
    D = copy.deepcopy(W)

    for k in range(vsize) :
        for i in range(vsize) :
            for j in range(vsize) :
                if (D[i][k] + D[k][j] < D[i][j]) :
                    D[i][j] = D[i][k] + D[k][j]
        printD(D)
```

- 복잡도
  - $O(n^3)$

알고리즘 테스트

```
C:₩WINDOWS₩system32₩cmd.exe
Shortest Path By Floyd's Algorithm
====================================
   0    7  INF  INF    3   10  INF
   7    0    4   10    2    6  INF
 INF    4    0    2  INF  INF  INF
 INF   10    2    0   11    9    4
   3    2  INF   11    0   13    5
  10    6  INF    9   13    0  INF
 INF  INF  INF    4    5  INF    0
====================================
   0    7   11   17    3   10  INF
   7    0    4   10    2    6  INF
  11    4    0    2    6   10  INF
  17   10    2    0   11    9    4
   3    2    6   11    0    8    5
  10    6   10    9    8    0  INF
 INF  INF  INF    4    5  INF    0
====================================
   0    7   11   13    3   10  INF
   7    0    4    6    2    6  INF
  11    4    0    2    6   10  INF
  13    6    2    0    8    9    4
   3    2    6    8    0    8    5
  10    6   10    9    8    0  INF
 INF  INF  INF    4    5  INF    0
====================================
   0    7   11   13    3   10   17
   7    0    4    6    2    6   10
  11    4    0    2    6   10    6
  13    6    2    0    8    9    4
   3    2    6    8    0    8    5
  10    6   10    9    8    0   13
  17   10    6    4    5   13    0
====================================
   0    5    9   11    3   10    8
   5    0    4    6    2    6    7
   9    4    0    2    6   10    6
  11    6    2    0    8    9    4
   3    2    6    8    0    8    5
  10    6   10    9    8    0   13
   8    7    6    4    5   13    0
====================================
   0    5    9   11    3   10    8
   5    0    4    6    2    6    7
   9    4    0    2    6   10    6
  11    6    2
   3    2    6
  10    6   10
   8    7    6
====================================
   0    5    9   11    3   10    8
   5    0    4    6    2    6    7
   9    4    0    2    6   10    6
  11    6    2    0    8    9    4
   3    2    6    8    0    8    5
  10    6   10    9    8    0   13
   8    7    6    4    5   13    0
계속하려면 아무 키나 누르십시오 . . .
```

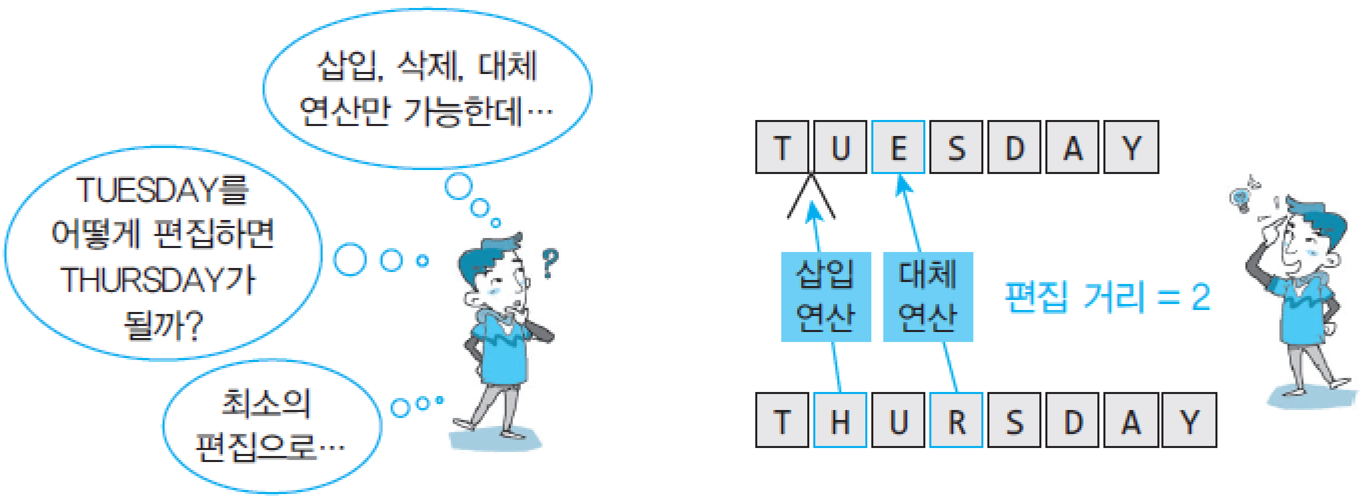


Dijkstra 알고리즘
(시작정점 0)의 결과와 동일

최종
A 행렬

7.7 편집 거리

두 문자열 S와 T가 주어졌고 S를 편집하여 T로 변환시키려고 한다. 사용할 수 있는 편집 연산은 한 문자의 삽입(insertion)이나 삭제(deletion), 그리고 대체(substitution)연산 뿐이다. S를 T로 변환시키는데 필요한 최소 편집 연산의 회수(편집 거리)를 구하라.



- 입력:
  - 길이가 각각 *m*과 *n*인 두 문자열 *S*과 *T*
  - $E_{ST}(m,n)$

순환 관계식

- 기반 상황
  - $E_{ST}(m, 0) = m$
  - $E_{ST}(0, n) = n$
- 일반 상황: $S[m], T[n]$
  - Case1: $S[m] = T[n]$
    - $E_{ST}(m, n) = E_{ST}(m-1, n-1)$
  - Case2: $S[m] != T[n]$
    - 대체 연산: $S[m]$을 $T[n]$ 으로 대체

      $E_{ST}(m-1, n-1)$ + 1
    - 삭제 연산: $S[m]$을 삭제

      $E_{ST}(m-1, n)$ + 1
    - 삽입 연산: T$[n]$을 S 에 삽입

      $E_{ST}(m, n-1)$ + 1

알고리즘

- 순환식

$$E_{ST}(m,n)=\begin{cases} n & m=0 \\ m & n=0 \\ E_{ST}(m-1,\ n-1) & S[m]=T[n] \\ 1+\min\left(E_{ST}(m-1,n-1),\ E_{ST}(m-1,n), E_{ST}(m,n-1)\ \right) & \text{otherwise} \end{cases}$$

- 알고리즘
  - 메모이제이션 사용

알고리즘 7.12 편집 거리(동적 계획법, 메모이제이션 사용)

```
def edit_distance_mem(S, T, m, n, mem):
    if m == 0: return n          # S가 공백이면, T의 모든 문자를 S에
    if n == 0: return m          # T가 공백이면, S의 모든 문자들을 식

    if mem[m-1][n-1] == None :   # 아직 해결되지 않은 문제이면
        if S[m-1]== T[n-1]:      # S와 T의 마지막 문자가 같으면,
            mem[m-1][n-1] = edit_distance_mem(S, T, m-1, n-1, mem)

        else:    # 그렇지 않으면, 세 연산을 모두 적용해 봄
            mem[m-1][n-1] = 1 + \
              min( edit_distance_mem(S, T, m, n-1, mem),
                   edit_distance_mem(S, T, m-1, n, mem),
                   edit_distance_mem(S, T, m-1, n-1, mem))
        # print("mem[%d][%d] = "%(m-1,n-1), mem[m-1][n-1])

    return mem[m-1][n-1]         # 해를 반환
```

- 복잡도
  - $O(mn)$

알고리즘 테스트

알고리즘 테스트 편집 거리(메모이제이션) 테스트

```
S = "tuesday"
T = "thursday"
m = len(S)
n = len(T)
print("문자열: ", S, T)
print("편집거리(분할정복    )= ", edit_distance(S, T, m, n))

mem = [[None for _ in range(n)] for _ in range(m)]
dist = edit_distance_mem(S, T, m, n, mem)
print("편집거리(메모이제이션)= ", edit_distance_mem(S, T, m, n, mem))
```

C:\WINDOWS\system32\cmd.exe

```
문자열:  tuesday thursday
편집거리(분할정복    )=  2
mem[0][0] =  0
mem[1][0] =  1
mem[2][0] =  2
mem[0][1] =  1
mem[1][1] =  1
mem[2][1] =  2
mem[1][2] =  1
mem[2][2] =  2
mem[0][2] =  2
mem[0][3] =  3
mem[1][3] =  2
mem[2][3] =  2
mem[3][4] =  2
mem[4][5] =  2
mem[5][6] =  2
mem[6][7] =  2
편집거리(메모이제이션)=  2
계속하려면 아무 키나 누르십시오 . . .
```

테이블이 채워지는 과정.

edit_distance_mem() 14행의 주석 한 줄을 제거하면 출력됨.

실습 과제

감사합니다!