동적계획법
(Dynamic Programming)

- 개요
- 피보나치 수열; 이항계수; 양의 정수 n의 합 표현 방법; 거스름돈 나누어주는 방법
- Weighted Interval Scheduling
- 연속하는 수들의 최대 합 구하기
- 그리드에서 경로 찾기
- LCS(Longest Common Subsequence) 문제

동적계획법

- 동적계획법은 주로 최적화 (optimization) 문제를 해결하는 데 주로 사용한다. 동적계획법을 이용한 문제 해결은 아래의 4 단계를 거친다.

단계 1. 최적 해의 재귀적 구조를 파악한다.

단계 2. 최적 해를 재귀적으로 (recursively) 구한다.
=> 최적해의 목적함수에 대한 점화식(재귀식)을 구한다.)

단계 3. 단계 2의 점화식(재귀식)으로부터 최적 해의 (목적함)수 값을 bottom-up(혹은 memoization)으로 구하면서 테이블에 저장한다. (bottom-up: 작은 부분문제에서 시작하여 큰 부분문제들까지 최적 해의 목적함수 값을 차례대로 구한다)

단계 4. 단계 3에서 구한 정보를 이용하여 최적 해를 찾는다.

※ 단계 1-3은 동적계획법으로 해를 구할 때 필수적인 과정임
단계 4는 최적 해의 목적함수 값만 구하는 경우는 필요 없고, 최적 해를 찾고자 하는 경우에 필요.

6. 연속하는 수들의 최대 합 구하기

- 문제: n개의 수 x_0, x_1, …, x_{n-1}에 대하여, 연속하는 수들의 최대 합을 구하라. (배열 num에 n 개의 수들이 저장되어 있다.)

방법 1 (비효율적 방법)

```
maxSum = -∞
for i = 0 to n-1
 for j = i to  n-1
// sum = num[i]부터 num[j]까지 합
     sum = 0
     for k = i to  j
        sum += num[k]
     if (maxSum < sum)
        ~~sum = maxSum~~
        maxSum = sum
```

시간복잡도: $O(n^3)$

방법 2 (비효율적 방법)

```
maxSum = -∞
for i = 0 to n-1
  sum = 0
  for j = i to n-1
     sum = sum + num[j]
     if (maxSum < sum)
        sum = maxSum
```

시간복잡도: $O(n^2)$

연속하는 수들의 최대 합 구하기 (계속)

- 문제: n개의 수 x_0, x_1, …, x_{n-1}에 대하여, 연속하는 수들의 최대 합을 구하라.
  (배열 num에 n 개의 수들이 저장되어 있다.)

방법 3: 분할과 정복 알고리즘

mid: 가운데 원소의 위치

배열의 원소들을 처음부터 mid까지 수들과 mid+1부터 마지막까지 수들로 나눈다.

왼쪽 반(처음부터 mid까지 수들)의 (최적)해를 구한다. 이 해를 S1이라 하자.

오른쪽 반(mid+1부터 마지막까지 수들)의 (최적)해를 구한다. 이 해를 S2라 하자.

중간에 걸쳐 있는(mid 좌우로 연속하는 수들)의 최적해를 구한다. 이 해를 S3이라 하자.

S1, S2, S3중 좋은 해가 주어진 입력에 대한 (최적)해이다.

연속하는 수들의 최대 합 구하기(계속)

- <u>문제:</u> n개의 수 $x_0, x_1, \ldots, x_{n-1}$에 대하여, 연속하는 수들의 최대 합을 구하라. (배열 num에 n 개의 수들이 저장되어 있다: x_i는 배열 num[i]에 저장되어 있음.)

방법 4 (효율적 방법) – 동적계획법

제약조건. $x_0 \ldots x_i$

(1) <u>**부분문제:**</u> $x_0, x_1, \ldots, x_i$에 대하여 x_i에서 끝나는 연속하는 수들의 최대 합을 구하라.

- 재귀적 해를 고안

(2) <u>**부분문제 (최적) 해의 목적함수**</u>

sum[i] = $x_0, x_1, \ldots, x_i$에 대하여 x_i에서 끝나는 연속하는 수들의 최대 합

(3) <u>**주어진 문제 (최적)해의 목적함수:**</u> $\max_{0 \le i \le n-1} \{sum[i]\}$

(4) <u>**부분문제 (최적)해의 목적함수에 대한 점화식(재귀식)**</u> (recurrence relation)

sum[i]= sum[i-1] + num[i], if sum[i-1] > 0
num[i]   , otherwise

- 시간복잡도: O(n)

sum[i] = sum[i-1] + num[i] (if sum[i-1] ≥ 0)
sum[i] = num[i] (if sum[i-1] < 0)

연속하는 수들의 최대 합 구하기

- 문제: n개의 수 $x_0, x_1, \ldots, x_{n-1}$에 대하여, 연속하는 수들의 최대 합을 구하라. (배열 num에 n 개의 수들이 저장되어 있다: x_i는 배열 num[i]에 저장되어 있음.)

방법 4 (효율적 방법) – 동적계획법

(1) 부분문제: $x_0, x_1, \ldots, x_i$에 대하여 x_i에서 끝나는 연속하는 수들의 최대 합을 구하라.
  - 재귀적 해를 고안

(2) 부분문제 (최적)해의 목적함수

sum[i] = $x_0, x_1, \ldots, x_i$에 대하여 x_i에서 끝나는 연속하는 수들의 최대 합

(3) 주어진 문제 (최적)해의 목적함수: $\max_{0 \le i \le n-1} \{sum[i]\}$

(4) 부분문제 (최적)해의 목적함수에 대한 점화식(재귀식)

$$sum[i] = \begin{cases} sum[i-1] + num[i], & \text{if } sum[i-1] > 0 \\ num[i], & \text{otherwise} \end{cases}$$

- 시간복잡도: O(n)

```
방법 4 (동적계획법)
sum, num은 배열(리스트)
sum[0] = num[0]
for i = 1 to n-1
   if(sum[i-1] > 0)
      sum[i] = sum[i-1]+num[i]
   else
      sum[i] = num[i]

sum 배열의 최대값을 출력

시간복잡도: O(n)
```

연속하는 수들의 최대 합 구하기

예: 4, -5, 7, -3, 6, -2, 9, -2, 4, -3, -2, 2, -3, -1, 2, 4

(1) 부분문제: $x_0, x_1, \ldots, x_i$에 대하여 x_i에서 끝나는 연속하는 수들의 최대 합을 구하라.

(2) 부분문제 해의 목적함수:

sum[i] = $x_0, x_1, \ldots, x_i$에 대하여 x_i에서 끝나는 연속하는 수들의 최대 합

(3) 주어진 문제의 해의 목적함수: $\max_{0 \le i \le n-1} \{sum[i]\}$

i	j
0	0 ~ n-1
1	1 ~ n-1
2	2 ~ n-1
⋮	

$O(n^2)$

(4) 부분문제 해의 목적함수에 대한 점화식(재귀식)

$$sum[i] = \begin{cases} sum[i-1] + num[i], & \text{if } sum[i-1] > 0 \\ num[i], & \text{otherwise} \end{cases}$$

* p[i]: $x_0, x_1, \ldots, x_i$에 대하여 x_i에서 끝나면서 합이 최대가 되는 연속하는 수들의 시작위치

i부터 j 까지의 합

$$p[i] = \begin{cases} p[i-1], & \text{if } sum[i-1] > 0 \\ i, & \text{otherwise} \end{cases}$$

i	0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
x_i: num[i]	4	-5	7	-3	6	-2	9	-2	4	-3	-2	2	-3	-1	2	4
sum[i]	4	-1	7	4	10	8	17	15	19	16	14	16	13	12	14	18
p[i]	0	0	2	2	2	2	2	2	2	2	2	2	2	2	2	2

연속하는 수들의 최대합 구하기 Python code

```python
def searchLinear(n):
    sum = [0 for i in range(n)]
    num = [4, -5, 7, -3, 6, -2, 9, -2, 4, -3, -2, 2, -3, -1, 2, 4]
    p = [0 for i in range(n)]

    for i in range(1, n):
        if (sum[i-1] >= 0):
            sum[i] = sum[i-1] + num[i]
            p[i] = p[i-1]
        else:
            sum[i] = num[i]
            p[i] = i

    print(p)
    print(sum)
    return max(sum)

print(searchLinear(16))
```

7. 그리드(Grid)에서 경로 찾기

- n × m 그리드의 각 셀 (i, j)에 양의 비용 C(i, j)가 주어져 있다. 가장 아래 행은 행 1이고, 가장 위의 행은 행 n이다. 각 셀 C(i, j) 로부터 한번에 갈 수 있는 셀들은 다음과 같다: 셀 (i+1, j −1) (j > 1인 경우)과 셀 (i+1, j) 및 셀 (i+1, j +1) (j < m인 경우).

(i+1, j-1) (if j>1)
(i+1, j), (i+1, j+1) (if j<m)

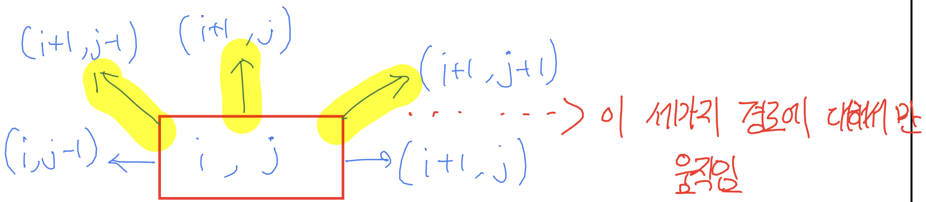
- 그리드의 가장 아래(bottom)에서 가장 위(top)로 가는 최소 비용의 경로를 찾아라. 여기서, 경로의 비용은 경로상에 있는 셀들의 비용 합이다.

- 예 4 × 5 grid

총합 12 (4,5)

2	**8**	**9**	**5**	**8**
4	4	6	2	3
5	7	5	6	1
3	2	5	4	8

(1,1)



Grid에서 경로 찾기

- **부분 문제**

  bottom에서 셀 (i,j)까지 가는 최소 비용의 경로를 찾아라.

- **부분문제의 최적 해 목적함수**

  A(i, j): bottom으로부터 셀 (i,j)가는 경로의 최소 비용

- **주어진 문제의 최적 해 값(목적함수)**

  $\min_{1 \le j \le m} A(n, j)$

- **부분문제 최적 해 값(목적함수)의 점화식(재귀식)**

Base case

$A(0, j) = 0 \quad \text{for } 1 \le j \le m$

혹은 $A(1, j) = C(1,j) \quad \text{for } 1 \le j \le m$

부분문제 최적해 값(목적함수)의 재귀적 정의

$$A(i,j) = \begin{cases} C(i,j) + \min\{A(i-1, j-1), A(i-1, j)\} & \text{if } 1 \le i \le n,\ j = m \\ C(i,j) + \min\{A(i-1, j), A(i-1, j+1)\} & \text{if } 1 \le i \le n,\ j = 1 \\ C(i,j) + \min\{A(i-1, j-1), A(i-1, j), A(i-1, j+1)\} & \text{if } 1 \le i \le n,\ j \ne 1 \text{ and } j \ne m \end{cases}$$

```python
# matrix= [[0 for i in range(m)] for j in range(n)]
import copy
m, n = map(int, input().split())
C = []
for i in range(m):
    C.append(list(map(int, input().split())))

print(C)

def grid(m, n, C):
    A = [0 for i in range(n)]
    for i in range(m-1, -1, -1):
        temp = copy.deepcopy(A) # 꼭 deepcopy 해주기!
        # 아래서 temp 값 변경될 때 A도 같은 주소 참조 중이라
        # temp 변경시 함께 변경 될 수 있음!!
        for j in range(n):
            if j == 0 and j < n-1:
                temp[j] = C[i][j] + min(A[j], A[j+1])
            elif j == n-1:
                temp[j] = C[i][j] + min(A[j-1], A[j])
            else:
                temp[j] = C[i][j] + min(A[j-1], A[j], A[j+1])
        print("pre:", temp)
        A = temp
        # print("after:",A)
    return min(A)

print(grid(m, n, C))

# 4 5
# 2 8 9 5 8
# 4 4 6 2 3
# 5 7 5 6 1
# 3 2 5 4 8
```

시간복잡도
O(mn)

Grid에서 경로 찾기

예:

2	**8**	**9**	**5**	**8**
4	4	6	2	3
5	7	5	6	1
3	2	5	4	8

배열 C

최소비용

경로를 찾으려면 어디서 왔는지를 보면 된다.

4	3	19	16	12	15
3	11	11	13	7	8
2	7	9	7	10	5
행1	3	2	5	4	8
행0	0	0	0	0	0

최소비용의 경로

굳이 초기화 안해도 됨. (관계없음)

$A(0,j)=0$ for $1 \le j \le m$

배열 A

수업시간에는 base case는 $A(1,j)=C(1,j)$ for $1 \le j \le m$

8. LCS(Longest Common Subsequence) 문제

- 문자열 *X* = *ABCBDAB*.
- *X*의 부분 수열(혹은 부분 서열: subsequence):

  *X*에서 0개 이상의 임의의 문자들을 삭제하여 얻을 수 있는 문자열
- 예:
  - **ABD, ABBB, BBDA**는 위 *X* 의 부분서열이다.
  - **AABB**는 위 *X*의 부분서열이 아니다.
- *LCS Problem*

두 문자열 X와 Y의 가장 긴 공통의 부분서열을 찾아라.

LCS 문제 (계속)

- X = $x_1x_2 \ldots x_m$, Y = $y_1y_2 \ldots y_n$
- 만약, X = ABCBDAB, Y = BDCABA라면 BCA는 부분서열이지만, LCS는 아니다.
- 그러면 LCS는 무엇인가?

LCS 문제 (계속)

- 두 DNA 서열의 유사도에 대한 여러 척도

  - LCS: 두 DNA 서열(두 문자열)의 유사도의 하나의 척도

- 예를 들어,

  X = ACCGGTCGACGT …

  Y = TTTCCTACTCGT …

가장 긴 공통의 부분 서열이 길면 두 DNA 서열이 유사하다고 말할 수 있다.

LCS 문제 (계속)

- LCS 문제

  X = $x_1x_2 \ldots x_m$과 Y = $y_1y_2 \ldots y_n$의 LCS를 구하라.

- 단순한 방법: X의 모든 부분서열에 대하여 이것이 Y의 부분서열인지를 조사한다

  => X의 부분서열의 개수는 2^m이므로 이 방법은 $\Omega(2^m)$의 시간이 걸린다.

LCS 문제 (계속)

- Subsequence 문제

수행시간

$X = x_1x_2 \ldots x_m$가 $Y = y_1y_2 \ldots y_n$의 subsequence인가?

$O(m+n)$

```
i =  j = 1
while (i <= m and j <= n)
      if (xi == yj )
         i += 1     (X의 위치 이동)
         j += 1     (Y의 위치 이동)
      else
         j += 1     (Y의 위치만 이동)

if (i > m)
   X is a subsequence of Y     (X는 Y의 부분서열이다)
else
   X is not a subsequence of Y     (X는 Y의 부분서열이 아니다)
```

LCS 문제 (계속)

- 부분문제
  $X_i = x_1 x_2 \dots x_i$와 $Y_j = y_1 y_2 \dots y_j$의 LCS를 구하라.

- 부분문제의 최적해 목적함수
  L(i,j): $X_i = x_1 x_2 \dots x_i$와 $Y_j = y_1 y_2 \dots y_n$의 LCS의 길이

- 주어진 문제의 최적해의 목적함수
  L(m,n)

- 부분문제의 최적해 목적함수 점화식(재귀식)
  L(i,j) = 0   if i = 0 or j = 0 // base case
  = L(i-1,j-1) + 1 if i>0 and j>0 and $x_i = y_j$
  = max{L(i,j-1), L(i-1,j)} if i>0 and j>0 and $x_i \neq y_j$ 같지 않은 경우 x_i가 포함 x
  or
  y_i가 포함 x

Example

X = ABCBDAB
Y = BDCABA

A ≠ B 둘중에 큰값

A = A 같을 경우 대각선 +1

	j	0	1	2	3	4	5	6
i			B	D	C	A	B	A
0		0	0	0	0	0	0	0
1	A	0	0	~~0~~	0	1	1	1
2	B	0	~~1~~	$\boxed{1}$	~~1~~	1	2	2
3	C	0	1	~~1~~	$\boxed{2}$	~~2~~	2	2
4	B	0	1	1	~~2~~	$\boxed{2}$	3	3
5	D	0	1	2	2	2	3	3
6	A	0	1	2	2	3	3	4
7	B	0	1	2	2	3	4	4

BDAB

두 문자열의 가장 길이가 긴 substring

X = ABCBDAB   Y = BDCABA

⇒ 동적 계획법.

두 문자열 P, T   (KMP) 알고리즘.

P: pattern   T: text

P가 T의 substring 인가? ⇒ Pattern matching 문제

T: This is a book

P: book

두 문자열 A, B를 align 하는 문제 (유사도)

A : abbc   B : babb

```
ab bc        abbc
 babb        babb
-1 +1 -1 +1 -1    -1 -1 +1 -1
-1           -2
```

대응하는 문자가 같으면 +1, 다르면 −1

〃 없으면 −1

대응하는 문자의 점수 총합이 가장 크게 align 하라.

```
# import copy
x = "%"+input()
y = "%"+input()

print(x, y)
print(len(x), len(y))

def LCS(x, y):
    C = [[0 for i in range(len(y))] for j in range(len(x))]
    L = [0 for i in range(len(y))]
    for i in range(len(x)):
        # temp = copy.deepcopy(L)
        for j in range(len(y)):
            if i == 0 or j == 0:
                C[i][j] = 0
            elif x[i] == y[j]:
                C[i][j] = C[i-1][j-1] + 1
            else:
                C[i][j] = max(C[i][j-1], C[i-1][j])
    print(C)
    return C[len(x)-1][len(y)-1]



print(LCS(x, y))
  %  B  D  C  A  B  A
[[0, 0, 0, 0, 0, 0, 0],
 [0, 0, 0, 0, 1, 1, 1],
 [0, 1, 1, 1, 1, 2, 2],
 [0, 1, 1, 2, 2, 2, 2],
 [0, 1, 1, 2, 2, 3, 3],
 [0, 1, 2, 2, 2, 3, 3],
 [0, 1, 2, 2, 3, 3, 4],
 [0, 1, 2, 2, 3, 4, 4]]

# ABCBDAB
# BDCABA
```