

# 설계 과제 #2

B989003 권덕재

## 1. 설계 과제 내용

Recursive version (rLCS), Recursive version + Memorization Technique (rmLCS), Dynamic Programming (dpLCS) 총 3개 버전의 LCS 알고리즘을 구현하고 입력 데이터에 대한 연산 횟수를 출력 및 그래프로 나타내어 비교하고 LCS path recovery(LCSPath), LCS 문제를 변형하여 "연속적으로 나타나는 최장 서브스트링" 구하기

## 2. 입력 데이터

입력 데이터 x, y 각 스트링의 집합은 길이 10 이하 3개, 10이상 20 이하 3개를 임의적으로 생성하였음. 최종 과제 시연에서 y 집합의 최대 길이를 20으로 하게 되면 시간이 너무 오래 걸리기 때문에 15 이하로 줄여서 테스트하였음.

```
xList = ["".join(chr(random.randint(ord('a'), ord('z')))) for _ in range(random.randint(5, 10))) for _ in range(3)]
yList = ["".join(chr(random.randint(ord('a'), ord('z')))) for _ in range(random.randint(10, 15))) for _ in range(3)]

for i in range(3):
    x, y = xList[i], yList[i]
    print(f'x{i + 1}: {x} / y{i + 1}: {y}')
```

x1: uvcujk / y1: tiwdsmqkbsq  
x2: jokvmwoqfd / y2: mzpdpipyij  
x3: lwtavbm / y3: vggppxtstgwgdt

## 3. 코드 설명

LCS 클래스 : 여러가지 버전의 LCS 알고리즘이 구현되어있는 클래스.

\_\_init\_\_(self) : 객체를 생성할 때 연산의 횟수를 의미하는 self.count 변수를 0으로 초기화하고 문자열 x, y와 x, y의 길이 lx, ly를 초기화하고 dp 배열 생성 및 초기화하는 메서드.

initCount(self) : self.count를 다시 0으로 초기화하는 메서드.

plusCount(self, count) : 연산이 발생할 때 연산 횟수만큼 self.count에 더하는 메서드.

popCount(self) : 연산의 횟수를 return 하고 다시 0으로 초기화하는 메서드.

setXY(self, x, y) : 문자열 쌍을 변경할 때마다 LCS 객체를 새롭게 생성하지 않도록 이미 생성되어 있는 객체의 self.x와 self.y를 변경하는 메서드.

rLCS(self, i, j) : 재귀적으로 실행하며 LCS를 구하는 메서드. 중복호출이 많아질 위험성이 있다.

rmLCS(self, i, j) : 재귀적으로 실행하는 점은 rLCS 메서드와 유사하지만 dp 배열을 사용하여 메모이제이션 방식을 사용하여 중복호출 문제점을 최소화한 메서드.

dpLCS(self) : dp 배열을 사용하여 iterative하게 LCS를 구하는 바텀업 방식을 사용한 메서드.

pathLCS(self) : LCS path recovery를 구하는 메서드. 화살표의 방향을 저장하는 배열을 따로 만들지 않고 dp[i][j]와 dp[i - 1][j], dp[i][j - 1] 중 어떤 값과 같은지에 따라 화살표가 아래쪽인지, 오른쪽인지 알 수 있으며 x[i]와 y[j]가 같다면 대각선 방향이라는 것을 알 수 있다.

dpLCSforConsecutiveSubstring(self) : 연속적으로 나타나는 최장 서브스트링을 구하기 위한 메서드. 연속적으로 나타나야 하기 때문에 x[i]와 y[j]가 다르다면 dp[i][j]를 0으로, 같다면 dp[i][j]를 dp[i - 1][j - 1] + 1로 초기화하는 방식을 사용한다. 즉 화살표의 방향은 대각선밖에 될 수 없음.

pathLCSforLongestConsecutiveSubstring(self) : 화살표의 방향이 대각선밖에 될 수 없으므로 x[i]와 y[j]가 다를 때까지 i와 j를 1씩 감소시킨다.

#### 4. 알고리즘 테스트

LCS 알고리즘이 모두 정상적으로 실행되는지 확인하기 위한 테스트 결과

```
lcsForTest = LCS("", "")

print('LCS 알고리즘이 정상적으로 잘 동작하는지 확인하기 위한 테스트 1')

x1, y1 = 'abcde', 'bde'
print('x1:', x1, '/ y1:', y1, '\n')

lcsForTest.setXY(x1, y1)

# rLCS Test
print('rLCS:', lcsForTest.rLCS(len(x1), len(y1)))
print('count:', lcsForTest.popCount())
print('pathLCS:', lcsForTest.pathLCS(), '\n')
lcsForTest.initDP()

# rmLCS Test
print('rmLCS:', lcsForTest.rmLCS(len(x1), len(y1)))
print('count:', lcsForTest.popCount())
print('pathLCS:', lcsForTest.pathLCS(), '\n')

# dpLCS Test
print('dpLCS:', lcsForTest.dpLCS())
print('count:', lcsForTest.popCount())
print('pathLCS:', lcsForTest.pathLCS(), '\n')

# getConsecutiveSubstring Test
print('consecutiveSubstring:', lcsForTest.pathLCSforLongestConsecutiveSubstring())
```

LCS 알고리즘이 정상적으로 잘 동작하는지 확인하기 위한 테스트 1  
x1: abcde / y1: bde

rLCS: 3  
count: 6  
pathLCS: bde

rmLCS: 3  
count: 6  
pathLCS: bde

dpLCS: 3  
count: 15  
pathLCS: bde

consecutiveSubstring: de

```
print('LCS 알고리즘이 정상적으로 잘 동작하는지 확인하기 위한 테스트 2')

x2, y2 = 'abcdefghijklm', 'nopqrstuvwxyz'
print('x2:', x2, '/ y2:', y2, '\n')

lcsForTest.setXY(x2, y2)

# rLCS Test
print('rLCS:', lcsForTest.rLCS(len(x2), len(y2)))
print('count:', lcsForTest.popCount())
print('pathLCS:', lcsForTest.pathLCS(), '\n')
lcsForTest.initDP()

# rmLCS Test
print('rmLCS:', lcsForTest.rmLCS(len(x2), len(y2)))
print('count:', lcsForTest.popCount())
print('pathLCS:', lcsForTest.pathLCS(), '\n')

# dpLCS Test
print('dpLCS:', lcsForTest.dpLCS())
print('count:', lcsForTest.popCount())
print('pathLCS:', lcsForTest.pathLCS(), '\n')

# getConsecutiveSubstring Test
print('consecutiveSubstring:', lcsForTest.pathLCSforLongestConsecutiveSubstring())
```

LCS 알고리즘이 정상적으로 잘 동작하는지 확인하기 위한 테스트 2  
x2: abcdefghijklm / y2: nopqrstuvwxyz

rLCS: 0  
count: 20801199  
pathLCS:

rmLCS: 0  
count: 339  
pathLCS:

dpLCS: 0  
count: 169  
pathLCS:

consecutiveSubstring:

모든 알고리즘이 정상적으로 작동하는 것을 확인할 수 있음.

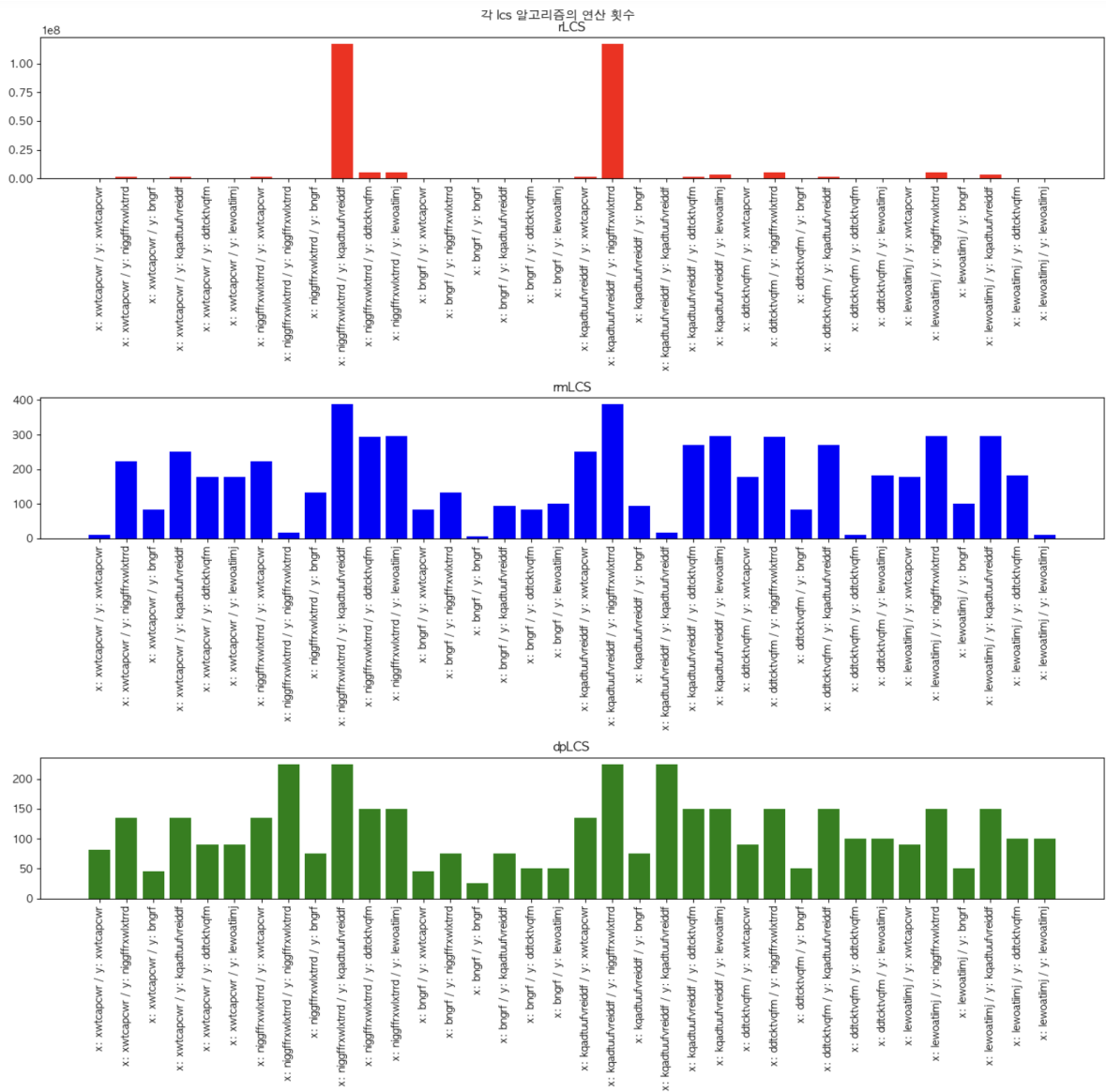
## 5. 시연 결과

### 1) 모든 문자열 쌍에 대한 각 알고리즘 별 연산 횟수 출력

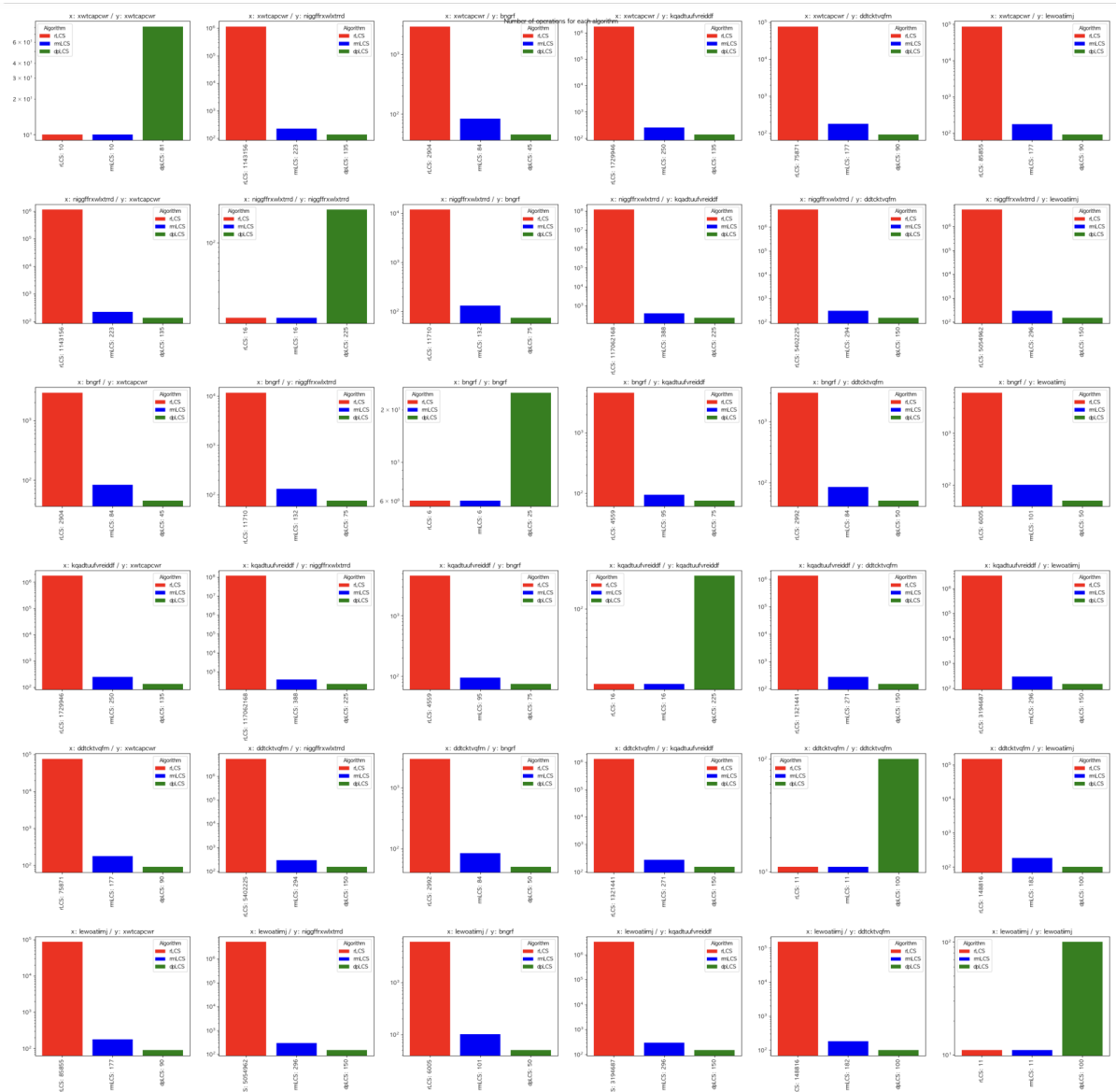
	x	y	rLCS	rmLCS	dpLCS	pathLCS
0	xwtcapcwr	xwtcapcwr	10	10	81	xwtcapcwr
1	xwtcapcwr	niggffrxwlxtrrd	1143156	223	135	xwtr
2	xwtcapcwr	bngrf	2904	84	45	r
3	xwtcapcwr	kqadtuufvreiddf	1729946	250	135	tr
4	xwtcapcwr	ddtcktvqfm	75871	177	90	tc
5	xwtcapcwr	lewoatiimj	85855	177	90	wt
6	niggffrxwlxtrrd	xwtcapcwr	1143156	223	135	xwtr
7	niggffrxwlxtrrd	niggffrxwlxtrrd	16	16	225	niggffrxwlxtrrd
8	niggffrxwlxtrrd	bngrf	11710	132	75	ngf
9	niggffrxwlxtrrd	kqadtuufvreiddf	117062168	388	225	frd
10	niggffrxwlxtrrd	ddtcktvqfm	5402225	294	150	f
11	niggffrxwlxtrrd	lewoatiimj	5054962	296	150	wt
12	bngrf	xwtcapcwr	2904	84	45	r
13	bngrf	niggffrxwlxtrrd	11710	132	75	ngr
14	bngrf	bngrf	6	6	25	bngrf
15	bngrf	kqadtuufvreiddf	4559	95	75	rf
16	bngrf	ddtcktvqfm	2992	84	50	f
17	bngrf	lewoatiimj	6005	101	50	
18	kqadtuufvreiddf	xwtcapcwr	1729946	250	135	ar
19	kqadtuufvreiddf	niggffrxwlxtrrd	117062168	388	225	frd
20	kqadtuufvreiddf	bngrf	4559	95	75	rf
21	kqadtuufvreiddf	kqadtuufvreiddf	16	16	225	kqadtuufvreiddf
22	kqadtuufvreiddf	ddtcktvqfm	1321441	271	150	dtvf
23	kqadtuufvreiddf	lewoatiimj	3194687	296	150	ati
24	ddtcktvqfm	xwtcapcwr	75871	177	90	tc
25	ddtcktvqfm	niggffrxwlxtrrd	5402225	294	150	d
26	ddtcktvqfm	bngrf	2992	84	50	f
27	ddtcktvqfm	kqadtuufvreiddf	1321441	271	150	dtvf
28	ddtcktvqfm	ddtcktvqfm	11	11	100	ddtcktvqfm
29	ddtcktvqfm	lewoatiimj	148816	182	100	tm
30	lewoatiimj	xwtcapcwr	85855	177	90	wa
31	lewoatiimj	niggffrxwlxtrrd	5054962	296	150	lt
32	lewoatiimj	bngrf	6005	101	50	
33	lewoatiimj	kqadtuufvreiddf	3194687	296	150	ati
34	lewoatiimj	ddtcktvqfm	148816	182	100	tm
35	lewoatiimj	lewoatiimj	11	11	100	lewoatiimj

(dpLCS의 경우 서로 같은 문자열 쌍에 대한 LCS를 구할 때 나머지 알고리즘보다 비효율적인 것을 확인할 수 있음.)

## 2) 각 알고리즘에 대한 모든 문자열 쌍의 LCS를 구하기 위한 연산 횟수 그래프



### 3) 모든 문자열 쌍에 대한 LCS 알고리즘의 연산 횟수 비교 그래프 (log scale 적용)



보너스) 모든 문자열 쌍에 대한 연속적으로 나타나는 최장 서브스트링

	x	y	longestConsecutiveSubstring
0	xwtcapcwr	xwtcapcwr	xwtcapcwr
1	xwtcapcwr	niggffrxwlxtrd	xw
2	xwtcapcwr	bngrf	r
3	xwtcapcwr	kqadtuufvreiddf	t
4	xwtcapcwr	ddtcktvqfm	tc
5	xwtcapcwr	lewoatiimj	w
6	niggffrxwlxtrd	xwtcapcwr	xw
7	niggffrxwlxtrd	niggffrxwlxtrd	niggffrxwlxtrd
8	niggffrxwlxtrd	bngrf	n
9	niggffrxwlxtrd	kqadtuufvreiddf	i
10	niggffrxwlxtrd	ddtcktvqfm	f
11	niggffrxwlxtrd	lewoatiimj	i
12	bngrf	xwtcapcwr	r
13	bngrf	niggffrxwlxtrd	n
14	bngrf	bngrf	bngrf
15	bngrf	kqadtuufvreiddf	r
16	bngrf	ddtcktvqfm	f
17	bngrf	lewoatiimj	
18	kqadtuufvreiddf	xwtcapcwr	a
19	kqadtuufvreiddf	niggffrxwlxtrd	d
20	kqadtuufvreiddf	bngrf	f
21	kqadtuufvreiddf	kqadtuufvreiddf	kqadtuufvreiddf
22	kqadtuufvreiddf	ddtcktvqfm	dt
23	kqadtuufvreiddf	lewoatiimj	a
24	ddtcktvqfm	xwtcapcwr	tc
25	ddtcktvqfm	niggffrxwlxtrd	d
26	ddtcktvqfm	bngrf	f
27	ddtcktvqfm	kqadtuufvreiddf	dd
28	ddtcktvqfm	ddtcktvqfm	ddtcktvqfm
29	ddtcktvqfm	lewoatiimj	t
30	lewoatiimj	xwtcapcwr	w
31	lewoatiimj	niggffrxwlxtrd	l
32	lewoatiimj	bngrf	
33	lewoatiimj	kqadtuufvreiddf	e
34	lewoatiimj	ddtcktvqfm	t
35	lewoatiimj	lewoatiimj	lewoatiimj

초기값:

$dp[i][0] = 0, 0 \leq i \leq \text{len}(x)$

$dp[0][j] = 0, 0 \leq j \leq \text{len}(y)$

관계식:

if  $(x[i-1] == y[j-1]) : dp[i][j] = dp[i-1][j-1] + 1$

else :  $dp[i][j] = 0$

## 6. 결과에 대한 생각

똑같은 문자열 쌍의 LCS를 구할 때를 제외하면 rLCS의 연산 횟수가 비교가 안 될 정도로 많은 것을 확인할 수 있습니다. 똑같은 문자열 쌍에 대한 결과가 이렇게 나온 이유는 dpLCS의 경우 어떤 경우든 그 문자열의 길이의 제곱만큼 dp 배열의 모든 부분을 탐색하지만 똑같은 문자열의 LCS를 재귀적으로 구하게 된다면 모든 경우에 i와 j가 1씩 감소하기 때문에 그 문자열의 길이만큼만 탐색하기 때문이라고 생각합니다.

## 7. AI에게 부분적으로 도움을 받은 부분

```
lcs = LCS("", "")
xyCounts = {'x':[], 'y':[], 'rLCS':[], 'rmLCS':[], 'dpLCS':[], 'pathLCS':[]}
for i in range(len(xList)):
    for j in range(len(yList)):
        x, y = xList[i], yList[j]
        lcs.setXY(x, y)
        xyCounts['x'].append(x)
        xyCounts['y'].append(y)
        lcs.rLCS(len(x), len(y))
        xyCounts['rLCS'].append(lcs.popCount())
        lcs.initDP()
        lcs.rmLCS(len(x), len(y))
        xyCounts['rmLCS'].append(lcs.popCount())
        lcs.dpLCS()
        xyCounts['dpLCS'].append(lcs.popCount())
        xyCounts['pathLCS'].append(lcs.pathLCS())
dfXY = pd.DataFrame(xyCounts)

yxCounts = {'x':[], 'y':[], 'rLCS':[], 'rmLCS':[], 'dpLCS':[], 'pathLCS':[]}
for i in range(len(xList)):
    for j in range(len(yList)):
        x, y = xList[i], yList[j]
        lcs.setXY(y, x)
        yxCounts['x'].append(y)
        yxCounts['y'].append(x)
        lcs.rLCS(len(y), len(x))
        yxCounts['rLCS'].append(lcs.popCount())
        lcs.initDP()
        lcs.rmLCS(len(y), len(x))
        yxCounts['rmLCS'].append(lcs.popCount())
        lcs.dpLCS()
        yxCounts['dpLCS'].append(lcs.popCount())
        yxCounts['pathLCS'].append(lcs.pathLCS())
dfYX = pd.DataFrame(yxCounts)

xxCounts = {'x':[], 'y':[], 'rLCS':[], 'rmLCS':[], 'dpLCS':[], 'pathLCS':[]}
for i in range(len(xList)):
    for j in range(len(xList)):
        x1, x2 = xList[i], xList[j]
        lcs.setXY(x1, x2)
        xxCounts['x'].append(x1)
        xxCounts['y'].append(x2)
        lcs.rLCS(len(x1), len(x2))
        xxCounts['rLCS'].append(lcs.popCount())
        lcs.initDP()
        lcs.rmLCS(len(x1), len(x2))
```

```

xxCounts['rmLCS'].append(lcs.popCount())
lcs.dpLCS()
xxCounts['dpLCS'].append(lcs.popCount())
xxCounts['pathLCS'].append(lcs.pathLCS())
dfXX = pd.DataFrame(xxCounts)

yyCounts = {'x':[], 'y':[], 'rLCS':[], 'rmLCS':[], 'dpLCS':[], 'pathLCS':[]}
for i in range(len(yList)):
    for j in range(len(yList)):
        y1, y2 = yList[i], yList[j]
        lcs.setXY(y1, y2)
        yyCounts['x'].append(y1)
        yyCounts['y'].append(y2)
        lcs.rLCS(len(y1), len(y2))
        yyCounts['rLCS'].append(lcs.popCount())
        lcs.initDP()
        lcs.rmLCS(len(y1), len(y2))
        yyCounts['rmLCS'].append(lcs.popCount())
        lcs.dpLCS()
        yyCounts['dpLCS'].append(lcs.popCount())
        yyCounts['pathLCS'].append(lcs.pathLCS())
dfYY = pd.DataFrame(yyCounts)

dfs = [dfXY, dfYX, dfXX, dfYY]
df = pd.concat(dfs).reset_index(drop=True)

```

위와 같이 길고 복잡한 코드를 Chat GPT에게 다음과 같은 명령을 통해 짧고 간결하게 축소

**덕재** 6개의 요소 중 2개를 선택해서 36개의 쌍을 만드는 방법이 무엇이 있을까?  
 이때 중복을 허용하며 선택한 두 요소의 순서가 뒤바뀔 수 있으며 1개의 원소를 2번 선택할 수 있다.  
 x1, x2, x3, y1, y2, y3 이렇게 6개의 요소를 사용하여 보아라.

**덕재** 6개의 요소 중 2개를 선택해서 36개의 쌍을 만드는 방법이 무엇이 있을까?  
 이때 중복을 허용하며 선택한 두 요소의 순서가 뒤바뀔 수 있으며 1개의 원소를 2번 선택할 수 있다.  
 x1, x2, x3, y1, y2, y3 이렇게 6개의 요소를 사용하여 보아라.

**덕재** 이 방법을 파이썬 코드로 구현할 수 있어?

python

 Copy code

```

import itertools

elements = ['x1', 'x2', 'x3', 'y1', 'y2', 'y3']
pairs = list(itertools.product(elements, repeat=2))

for pair in pairs:
    print(pair)


```



덕재

## 더 짧게 줄일 수 있을까?

python

 Copy code

```
elements = ['x1', 'x2', 'x3', 'y1', 'y2', 'y3']
pairs = [(x, y) for x in elements for y in elements]

for pair in pairs:
    print(pair)
```

```
for i in range(3):
    x, y = xList[i], yList[i]
    elements.append(x)
    elements.append(y)
pairsOfStrs = [(x, y) for x in elements for y in elements]
operateCounts = {'x':[], 'y':[], 'rLCS':[], 'rmLCS':[], 'dpLCS':[], 'pathLCS':{}}
for x, y in pairsOfStrs:
    lcs.setXY(x, y)
    operateCounts['x'].append(x)
    operateCounts['y'].append(y)
    lcs.rLCS(len(x), len(y))
    operateCounts['rLCS'].append(lcs.popCount())
    lcs.initDP()
    lcs.rmLCS(len(x), len(y))
    operateCounts['rmLCS'].append(lcs.popCount())
    lcs.dpLCS()
    operateCounts['dpLCS'].append(lcs.popCount())
    operateCounts['pathLCS'].append(lcs.pathLCS())
df = pd.DataFrame(operateCounts)
```

### 8. 느낌점

AI의 코딩 능력을 활용하여 코드를 훨씬 더 간결하게 수정할 수 있었고 앞으로 미래에는 코딩 능력 뿐만 아니라 AI에게 원하는 것을 명령하는 능력 또한 중요해질 것 같다는 생각이 들었습니다. 또한 dpLCS의 경우 서로 같은 문자열에 대한 LCS를 구할 때 rLCS와 rmLCS보다 월등히 많은 연산이 필요한 것처럼 데이터의 유형마다 적용했을 때 유리한 알고리즘이 따로 있다는 것을 다시 한 번 더 깨닫게 되었고 앞으로 실전에서도 이와 같이 적절한 알고리즘을 사용하여 효율적인 개발자가 되고 싶다는 생각이 들었습니다.