# 자료구조.2021

# 과제 1

20172674
신동녘

# 다항식 – 방식1

```
수식 1을 입력하세요: 3 0 6 3
수식 2를 입력하세요: 7 5 1
수식 1+2 는 3 * x^3 + 7 * x^2 + 11 * x^1 + 4
수식 1*2 는 21 * x^5 + 15 * x^4 + 45 * x^3 + 51 * x^2 + 21 * x^1 + 3

수식에 값을 넣으세요(ex: 1 1) 1 1
결과값은 12
수식에 값을 넣으세요(ex: 1 1) 3 2
결과값은 78
수식에 값을 넣으세요(ex: 1 1) 4 -1
결과값은 -18
수식에 값을 넣으세요(ex: 1 1)
```

*(실행결과 캡처화면)*

먼저 다항식의 저장 방식 중 첫번째 방식은 고차 순으로 모든 항(계수가 0인 항 포함)의 계수를 배열에 넣어 관리하는 방식입니다. 수식 두 개를 입력 받아, 이 두 식에 대한 덧셈과 곱셈의 식을 계산합니다. 이렇게 정리된 네 가지 식 중 한 가지 식에 사용자 입력값을 대입하여 결과값을 구하는 문제입니다.

```python
# 두 다항식을 더하는 함수
def poly_add(a, b):
    z = []# 결과를 담을 다항식 선언
    apos = bpos = 0# 배열(다항식)을 순차적으로 탐색하기 위한 인덱스
    degree_a = a.degree# 다항식 a의 차수
    degree_b = b.degree# 다항식 b의 차수

    while(degree_a > 0) and (degree_b > 0):
        # a가 차수가 더 높다면
        if degree_a > degree_b:
            z.append(a.coef[apos])
            apos += 1
            degree_a -= 1
        # a와 b가 차수가 같다면
        elif degree_a == degree_b:
            z.append(a.coef[apos] + b.coef[bpos])
            apos += 1
            bpos += 1
            degree_a -= 1
            degree_b -= 1
        # b가 차수가 높다면
        else:
            z.append(b.coef[bpos])
            bpos += 1
            degree_b -= 1
    return polynomial(z)
```

덧셈의 경우 apos와 bpos에 해당하는 인덱스 관리 변수를 생성하여 두 식의 차수가 더 큰 쪽이 있다면 자동으로 결과 배열에 추가시키고, 두 식의 차수가 같다면 더한 값을 결과 배열(사진 상 z에 해당함)에 추가시키는 방식으로 진행하였습니다. 이 때 배열로 다항식을 관리하는 방식에 따라 계수가 0이더라도 차수가 존재한다면 칸을 차지하고 있고 이에 따라 결과 배열은 빈틈없이 자리가 두 다항식의 최고 차수에 맞게 채워질 수 있습니다.

이 때 기존에 주어진 poly_add 함수의 while문의 반복 조건은 동차 다항식 두 개가 주어질 경우 올바른 계산이 이루어지지 않았기 때문에 두 차수가 모두 상수항을 가리킬 때로 설정하였다는 점에서 어느 다항식의 경우에든 올바르게 작동할 수 있도록 하였습니다.

곱셈의 경우 방식1이 계수가 없는 차수에도 0을 채워 넣는 다는 점에서 곱셈 결과 최고차 항으로부터 상수항까지 담을 수 있는 길이의 0 배열을 생성하였고, 이후 모든 항을 단순 교차 연산하여 i+j 즉 곱했을 때의 차수에 맞는 칸에 곱셈 결과를 더해 주어 결과를 도출하였습니다.

```python
# 두 다항식을 곱하는 함수
def poly_mult(a, b):
    degree_a = a.degree
    degree_b = b.degree
    # 곱해서 생성되는 다항식은 polymonial class의 정의 상 두 차수 합의 -1이다.
    z = [0]*(degree_a+degree_b-1)

    for i in range(degree_a):
        for j in range(degree_b):
            z[i+j] += a.coef[i]*b.coef[j]

    return polynomial(z)
```

# 다항식 – 방식2

```
수식 1을 입력하세요: 3 3 6 1 3 0
수식 2를 입력하세요: 7 2 5 1 1 0
수식 1+2 는 3 * x^3 + 7 * x^2 + 11 * x^1 + 4
수식 1*2 는 21 * x^5 + 15 * x^4 + 45 * x^3 + 51 * x^2 + 21 * x^1 + 3

수식에 값을 넣으세요(ex: 1 1) 1 1
결과값은 12
수식에 값을 넣으세요(ex: 1 1) 3 2
결과값은 78
수식에 값을 넣으세요(ex: 1 1) 4 -1
결과값은 -18
수식에 값을 넣으세요(ex: 1 1)
```

*(실행결과 캡처화면)*

다음 다항식의 저장 방식 중 두번째 방식은 고차 순으로 항(계수가 0인 항은 포함 포함 하지 않고)의 계수와 차수를 동시에 배열에 넣어 관리하는 방식입니다. 수식 두 개를 입력 받아, 이 두 식에 대한 덧셈과 곱셈의 식을 계산합니다. 이렇게 정리된 네 가지 식 중 한 가지 식에 사용자 입력값을 대입하여 결과값을 구하는 문제입니다.

```python
# 두 다항식을 더하는 함수
def poly_add(a, b):
    z = []  # 결과를 담을 다항식 선언
    apos = bpos = 0  # 배열(다항식)을 순차적으로 탐색하기 위한 인덱스
    degree_a = a.degree  # 다항식 a의 차수
    degree_b = b.degree  # 다항식 b의 차수

    while(apos < degree_a) and (bpos < degree_b):
        # a가 차수가 더 높다면
        if a.coef_deg[apos][1] > b.coef_deg[bpos][1]:
            z.append([a.coef_deg[apos][0], a.coef_deg[apos][1]])
            apos += 1
        # a와 b가 차수가 같다면
        elif a.coef_deg[apos][1] == b.coef_deg[bpos][1]:
            z.append([a.coef_deg[apos][0] + b.coef_deg[bpos][0], a.coef_deg[apos][1]])
            apos += 1
            bpos += 1
        # b가 차수가 높다면
        else:
            z.append([b.coef_deg[bpos][0], b.coef_deg[bpos][1]])
            bpos += 1
    return polynomial(z)
```

방식2의 경우 방식1처럼 차수 자체를 기록하는 변수를 따로 만들 필요가 없었습니다. 각 계수는 각자의 차수를 담고 있으므로 차수에 직접 접근하여 최고 차수부터 순차적으로 탐색하는 방식으로 조건문에 관한 판단을 진행할 수 있었습니다. 두 다항식의 차수를 비교하며 공통된 항을 가지고 있을 때에만 더해주고 그렇지 않은 경우 더 고차의 항을 가지고 있는 쪽만 결과 배열에 옮겨 담았습니다.

```python
def poly_mult(a, b):
    z = []  # 곱셈 결과를 담을 리스트 z
    z_list = []  # z가 가지고 있는 차수를 담을 리스트
    degree_a = a.degree  # a의 차수
    degree_b = b.degree  # b의 차수

    for i in range(degree_a):
        for j in range(degree_b):
            # 임시로 곱셈 결과를 해당 다항식의 방식대로 저장
            temp = [a.coef_deg[i][0]*b.coef_deg[j][0], a.coef_deg[i][1]+b.coef_deg[j][1]]
            if temp[1] in z_list:  # 같은 차수가 이미 있다면
                for k in range(len(z)):
                    # 해당 차수의 계수를 증가시켜준다.
                    if z[k][1] == temp[1]:
                        z[k][0] += temp[0]
            else:  # 같은 차수가 없다면
                z.append(temp)  # 임시 다항식을 z에 추가해주고
                z_list.append(temp[1])  # 해당 차수가 있음을 리스트에 추가하여 표기한다.

    return polynomial(z)
```

곱셈의 경우 방식1처럼 간단하게 이루어지지 않았습니다. 가장 먼저 앞선 방식과 마찬가지로 이중 배열을 통해 각 행에 접근합니다. 이 때 임시로 곱셈 결과가 되는 계수와 차수로 이루어진 배열을 만들어 두었습니다. 또 z_list라는 z라는 결과 배열에 담긴 차수를 모아두는 배열을 만들어두어 임시 배열과 비교할 수 있도록 하였고 해당 차수가 이미 존재한다면 해당 차수 위치의 계수를 증가시키고, 없다면 임시로 만들어 둔 배열 자체를 임시 배열에 추가 및 차수를 저장하는 z_list배열에도 차수를 저장해 주었습니다.

```
# 수식 출력 후 계산 부분
ploy_list = [a, b, c, d]# 다항식을 한 데에 묶어 효과적으로 관리하기 위한 리스트 생성
while (True):
    print("수식에 값을 넣으세요(ex: 1 1) ", end='')
    n, m = map(int, input().split())
    result = 0
    if n >= 1 and n <= 4:
        result = ploy_list[n-1].calc_poly(m)
        print("결과값은 %d" % (result))
    else:
        print("입력이 잘못되었습니다.")
```

마지막으로 두 방식 모두에 해당 수식에 값을 대입하여 결과값을 얻을 때, 다항식 네 개를 리스트로 묶어 간단하게 관리할 수 있도록 하였습니다.

## 행렬 – 방식1(좌), 방식2(우)

```
행렬의 규격을 입력하세요. 3
행렬 1의 데이터를 입력하세요. 1 0 3 0 0 0 2 0 0
행렬 2의 데이터를 입력하세요. 2 0 1 0 2 0 0 1 0


방식1

행렬1(9)
1 0 3
0 0 0
2 0 0

행렬2(9)
2 0 1
0 2 0
0 1 0

행렬 1+2(9)
3 0 4
0 2 0
2 1 0

행렬 1*2(9)
2 3 1
0 0 0
4 0 2
```

```
방식2

행렬3(9)
0 0 1
0 2 3
2 0 2

행렬4(12)
0 0 2
0 2 1
1 1 2
2 1 1

행렬 3+4(15)
0 0 3
0 2 4
1 1 2
2 0 2
2 1 1

행렬 3*4(15)
0 0 2
0 2 1
0 1 3
2 0 4
2 2 2

******
```

*(실행결과 캡처화면)*

행렬의 규격과 두 행렬에 대한 데이터를 입력 받고, 이에 대해 2차원 배열로 저장하거나, 희소 행렬방식으로 저장하고, 각각의 방식에 대한 행렬의 덧셈과 곱셈을 진행하여 데이터 관리의 효율성을 확인하는 문제입니다.

```
# 2차원 배열 행렬 덧셈 함수
def add_normal(a, b, n):
    c = [[0]*n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            c[i][j] = a[i][j] + b[i][j]
    return c


# 2차원 배열 행렬 곱셈 함수
def multiply_normal(a, b, n):
    c = [[0] * n for _ in range(n)]
    for i in range(n):
        for j in range(n):
            for k in range(n):
                c[i][j] += a[i][k] * b[k][j]
    return c
```

먼저 2차원 배열로 저장된 행렬의 덧셈과 곱셈은 경우의 수에 관한 고려 없이 각각의 위치에 알맞는 연산을 진행하였습니다. 특히 덧셈의 경우 c라는 결과 행렬을 입력된 규격에 맞게 0으로 초기화하여 각각의 위치에 해당하는 두 행렬의 원소를 더해주었습니다. 곱셈은 일반 행렬 연산과 같이 모든 원소에 대해 c[i][j]원소에 대해서 각 위치에 누적되어야하는 행렬의 곱들의 위치를 반복문에 따라 알맞게 설정해주었습니다. 예컨데 a와 b의 행렬 곱에서 c[i][j]위치에 누적되는 곱은 (a[i][0]*b[0][j])+(a[i][1]*b[1][j])+(a[i][2]*b[2][j])+…+(a[i][n]*b[n][j])에 해당합니다.

희소행렬의 덧셈은 행렬을 저장한 배열 상의 좌표값과 실제 행렬에서의 좌표값의 차이가 있다는 점에서 2차원 배열에 저장하는 행렬과 그 계산 방식에서 궤를 달리합니다. 따라서 행렬 a와 b의 위치를 정해두고 각각을 순차 탐색하면서 row와 col의 대소 비교를 통해 완전히 좌표가 같은 경우에만 두 수를 더한 값과 좌표를 하나의 배열로 만들어 결과 배열인 z에 추가합니다. 그 외에 값에서 차이가 날 경우 좌표값이 더 작은 행렬을 가진 값먼저 마찬가지로 좌표값과 해당 값으로 이루어진 임의의 배열을 만들어 결과 배열인 z에 추가하였습니다. 이 때 while문의 반복 조건은 어느 한 쪽 행렬이라도 마지막에 이르면 종료되는 것으로 설정하였으므로 마지막부분에 상대적으로 다른 행렬에 비해 길이가 길어 처리되지 못하고 남아있는 부분에 대해 처리하는 부분을 추가하였습니다.

```python
# 서로 다른 두 희소행렬을 더하는 함수
def add_sparse(a, b):
    z = []
    apos = bpos = 0 # a,b 행렬 각각 현재 처리되고 있는 위치를 가리키는 변수

    while (apos < len(a)) and (bpos < len(b)):
        # 겹치지 않은 부분은 차례대로 옮겨주고
        if a[apos][0] < b[bpos][0]:
            z.append([a[apos][0], a[apos][1], a[apos][2]])
            apos += 1
        elif a[apos][0] == b[bpos][0]:
            if a[apos][1] < b[bpos][1]:
                z.append([a[apos][0], a[apos][1], a[apos][2]])
                apos += 1
            elif a[apos][1] == b[bpos][1]: # 완벽하게 행렬의 위치가 같은 경우 더한 값을 z에 추가해준다.
                z.append([a[apos][0], a[apos][1], a[apos][2] + b[bpos][2]])
                apos += 1
                bpos += 1
            else:
                z.append([b[bpos][0], b[bpos][1], b[bpos][2]])
                bpos += 1
        else:
            z.append([b[bpos][0], b[bpos][1], b[bpos][2]])
            bpos += 1

    # 상대적으로 행렬이 다른 행렬에 비해 길어 처리되지 못한 나머지 부분을 처리하는 부분
    while apos < len(a):
        z.append([a[apos][0], a[apos][1], a[apos][2]])
        apos += 1
    while bpos < len(b):
        z.append([b[bpos][0], b[bpos][1], b[bpos][2]])
        bpos += 1

    return z
```

```python
# 서로 다른 두 희소행렬을 곱하는 함수
def multiply_sparse(a, b):
    z = []
    for i in range(len(a)):
        for j in range(len(b)):
            if a[i][1] == b[j][0]: # 위치가 같아야만 곱셈이 가능
                isAppended = False
                for k in range(len(z)):
                    if z[k][0] == a[i][0] and z[k][1] == b[j][1]: # 해당 인덱스가 존재하야 존재하는 인덱스에 곱셈 결과 누적
                        z[k][2] += a[i][2]*b[j][2]
                        isAppended = True
                if not isAppended:
                    z.append([a[i][0], b[j][1], a[i][2]*b[j][2]]) # 해당 인덱스가 존재하지 않아 새로 추가
    return z
```

마지막으로 희소행렬의 곱셈입니다. 희소행렬은 덧셈과 마찬가지로 모든 좌표에 대한 모든 좌표의 곱을 수행하는 과정에서 행렬의 곱 연산이 실제 이루어지는 부분에 관해서만 처리를 해야하고 여러 곱 연산이 더해져 최종 결과 배열의 한 좌표를 이루게 된다는 점에서 까다로웠습니다. 먼저 앞서 언급한 것처럼 조건 문에서 연산 하는 행렬의 Y좌표와 피연산 행렬의 X좌표가 같은지를 판단합니다. 이 때 isAppended 변수는 곱해져서 만들어진 좌표에 값이 존재하지 않을 경우 새로 행렬의 좌표와 값을 곱해주기 위함입니다. 이때 연산 행렬의 Y좌표와 피연산 행렬의 X좌표가 같고 두 항 모두가 0이 아니라면 곱해진 값은 (연산 행렬의 X좌표, 피연산행렬의 Y좌표) 위치에 누적된다는 아이디어를 활용하여 코드를 완성하였습니다.

```python
# 데이터를 입력받아 2차원 배열로 행렬로 만드는 함수
def normal_matrix(lst, n):
    index = 0
    z = []
    # 정해진 행렬 규격에 맞게 한 행을 각각의 리스트로 만든다.
    for _ in range(n):
        temp = []
        for _ in range(n):
            temp.append(lst[index])
            index += 1
        z.append(temp)

    return z
```

```python
# 2차원 배열 행렬을 희소행렬로 바꾸는 함수
def sparse_matrix(matrix, n):
    z = []
    for i in range(n):
        for j in range(n):
            if matrix[i][j] != 0:
                z.append([i, j, matrix[i][j]])
    return z
```

*데이터를 2차원 배열 행렬로 변환하는 함수(좌), 희소행렬로 변환하는 함수(우)*

마지막으로 데이터를 입력 받아 2차원 배열 형태의 행렬을 만드는 함수와 그 데이터를 희소 행렬로 변화하는 함수 입니다. 먼저 2차원 배열 형태로 생성하는 normal_matrix 함수는 데이터 리스트와 규격을 함께 매개변수로 받아, 규격만큼의 반복문을 통해 한 행의 횟수만큼의 리스트를 만들고 해당 행을 결과 배열에 추가하는 방식으로 진행하여 N X N 행렬을 완성할 수 있었습니다. 다음 희소행렬을 만드는 sparse_matrix 함수는 입력받은 행렬을 규격에 따라 이중 for문으로 순회하면서 데이터가 있을 경우에만 [행, 열, 값] 으로 이루어진 리스트를 생성하여 결과 배열에 추가하였습니다.

**감사합니다.**