<문제은행 [2] _ 클래스화>

2. 1, 1, 2, 3, 5, 8, 13, 21, 34, 55, 89, ... 일명 피보나치 수열의 20번째 항을 구하도록 프로그램 해보자!
3. 1, 1, 1, 2, 3, 4, 6, 9, 13, 19, 28, 41, 60, 88, 129, ...
   이와 같은 숫자의 규칙을 찾아 25번째 항을 구하도록 프로그램 해보자!

[MyCustomSequenceTest (main) - MyCustomSequence - initialSet]

- MyCustomSequenceTest (main)

```
int[] arr = { 1, 1 };

MyCustomSequence mcs = new MyCustomSequence(arr,  bias: 0);
System.out.println("20번째 데이터 = " + mcs.getNthOrderData( count: 20));

int[] arr2 = { 1, 1, 1, 2 };

MyCustomSequence mcs2 = new MyCustomSequence(arr2,  bias: 1);
System.out.println("25번째 데이터 = " + mcs2.getNthOrderData( count: 25));

MyCustomSequence mcs3 = new MyCustomSequence(arr,  bias: 0,  jumping: 0);
System.out.println("20번째 데이터 = " + mcs3.getNthOrderData( count: 20));

int[] arr3 = { 1, 1, 1 };

MyCustomSequence mcs4 = new MyCustomSequence(arr3,  bias: 0,  jumping: 1);
System.out.println("25번째 데이터 = " + mcs4.getNthOrderData( count: 25));
```

- 각기 다른 초기값을 가지는 배열 3가지 arr, arr2, arr3 선언

- 생성자를 통해 arr 배열과 bias값을 넘겨주며 MyCustomSequence 형식의
  객체 mcs를 생성하고, getNthOrderData 메소드를 호출해 20번째 데이터를 출력한다.

- 생성자를 통해 arr2 배열과 bias값을 넘겨주며 MyCustomSequence 형식의
  객체 mcs2를 생성하고, getNthOrderData 메소드를 호출해 25번째 데이터를 출력한다.

- 생성자를 통해 arr 배열과 bias값, jumping값을 넘겨주며 MyCustomSequence 형식의
  객체 mcs3를 생성하고, getNthOrderData 메소드를 호출해 20번째 데이터를 출력한다.

- 생성자를 통해 arr 배열과 bias값, jumping값을 넘겨주며 MyCustomSequence 형식의
  객체 mcs4를 생성하고, getNthOrderData 메소드를 호출해 25번째 데이터를 출력한다.

- MyCustomSequence

```
public class MyCustomSequence {
    InitialSet is;

    public MyCustomSequence (final int[] arr, int bias) { is = new InitialSet(arr, bias); }
    public MyCustomSequence (final int[] arr, int bias, int jumping) { is = new InitialSet(arr, bias, jumping); }

    public int getNthOrderData (int count) { return is.getNthOrderData(count); }
}
```

1. InitialSet 형식의 객체 is를 선언한다.

2. MyCustomSequence 생성자[1]
 : arr 배열과 bias를 main으로부터 받아와
 is 객체의 생성자를 호출하며 이 값들을 넘겨준다.

3. MyCustomSequence 생성자[2]
 : arr 배열과 bias, jumping을 main으로부터 받아와
 is 객체의 생성자를 호출하며 이 값들을 넘겨준다.

4. getNthOrderData
 : main으로부터 count 값을 받아와 is 객체의 getNthOrderData 메소드를 호출하며 count값을 넘겨준다. 그리고 getNthOrderData 메소드의 결과값을 반환한다.

- **initialSet**

1. 변수 선언

```
// 정수형 배열, 길이, 보정치(BIAS), 점핑
int[] initArr;
int length;
int bias;
int jumping;

int[] seqArr;
```

정수형 배열인 initArr
배열의 길이를 저장할 변수인 length
보정치 bias: 순차적으로 더하다가 연결되지 않는 경우
Jumping: 순차적으로 더하지 않고 몇 개 건너뛰고 처리하는 경우
계산된 값들이 저장될 배열 seqArr

2. initialSet 생성자

- 생성자 [1]

```
// 1번 케이스
public InitialSet (final int[] arr, int bias) {
    length = arr.length;

    // 배열 내용 복사
    this.initArr = arr.clone();
    this.bias = bias;

    jumping = 0;
}
```

main으로부터 전달받은 arr의 길이를 구해 length에 저장한다.
arr.clone()을 사용해 전달받은 arr 배열을 복사해 initArr 배열에 대입한다.
main으로부터 전달받은 bias 값을 **initialSet** 클래스의 bias에 대입한다.
jumping 값을 0으로 초기화한다.

- 생성자 [2]

```
// 2번 케이스
public InitialSet (final int[] arr, int bias, int jumping) {
    length = arr.length;

    this.initArr = arr.clone();
    this.bias = bias;
    this.jumping = jumping;
}
```

main으로부터 전달받은 arr의 길이를 구해 length에 저장한다.
arr.clone()을 사용해 전달받은 arr 배열을 복사해 initArr 배열에 대입한다.
main으로부터 전달받은 bias 값을 initialSet 클래스의 bias에 대입한다.
main으로부터 전달받은 jumping 값을 initialSet 클래스의 jumping에 대입한다.

3. getNthOrderData

```
public int getNthOrderData (int count) {
    seqArr = new int[count];

    initArr();
```

main에서 생성자를 호출한 다음 수행되는 메소드로 몇 번째 데이터까지의 값을 구할지 결정하는 count 값을 받아온다.
count만큼의 크기를 가지는 seqArr 배열을 생성하고 initArr 메소드를 호출해
전달받은 arr 배열에서 항의 값이 정해진 인덱스까지만 initArr 배열에 복사한다.
(값이 정해진 항은 다시 계산할 필요가 없기 때문에)

- initArr

```
public void initArr () {
    for (int i = 0; i < length; i++) {
        seqArr[i] = initArr[i];
    }
}
```

seqArr배열의 각 인덱스값을 initArr배열의 인덱스에 순차적으로 대입한다.
이는 for문을 통해 length만큼 반복된다.
-> 항의 값이 정해진 인덱스까지 복사
1) arr = {1,1}, count 20, bias 0, jumping 0
- initArr( ) -> seqArr = {1,1}

```
int tmp;
// 2 ~ 19 or 4 ~ 24 or 3 ~ 24
for (int i = length; i < count; i++) {
    tmp = 0;
    // 0 ~ 1 or 0 ~ 3 <<<--- bias
    // 우리가 맞춰야 하는 조건은 [0] + [1] + [2] = [4]
    //                        [1] + [2] + [3] = [5]
    //                        [2] + [3] + [4] = [6]

    // 앞서서는 길이값이 3인 상태에서 실제 배치가 인덱스 [4]에 하므로
    // 길이 + 보정치로 인덱스 위치를 맞추는 작업이 필요했기 때문에 사용함

    // 현재 케이스는 길이가 애초에 4이므로 인덱스 [4]는 맞춰짐
    // 그러나 연산을 3번만 해야하므로 bias를 반복 계산을 제한하는 루틴으로 활용함
```

임시합을 담을 tmp 변수를 선언한다.
항의 값이 정해진 인덱스 이후의 항부터 수열의 규칙에 따라 계산하기 위해
for문을 통해 length부터 count까지 반복을 수행한다.
tmp의 값을 0으로 초기화한다.

```
        for (int j = length; j > bias; j--) {
            // 4 - 4, 4 - 3, 4 - 2 ===> 0, 1, 2
            // 5 - 4, 5 - 3, 5 - 2 ===> 1, 2, 3

            // 2번째 방식의 jumping을 이용하는 경우는
            // [0] + [2] = [3]
            // [1] + [3] = [4]
            tmp += seqArr[i - j];
            j -= jumping;
        }

        seqArr[i] = tmp;
        System.out.printf("seqArr[%d] = %d\n", i, seqArr[i]);
    }

    return seqArr[count - 1];
}
```

bias는 j 반복문 안에서 반복계산을 제한하는 변수
I 반복문과 j 반복문을 통해 원하는 항인 count항까지의 배열의 값을 계산하고,
seqArr[count -1] 의 값을 반환한다.
(ex) 20번째 데이터는 19번 인덱스를 의미하기 때문에)

- 계산 과정 1,3

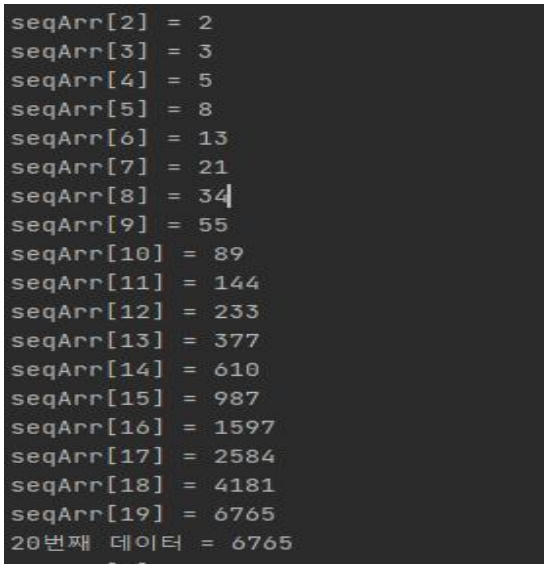
initArr -> seqArr = {1,1}
arr = {1,1}, count 25, bias 0, jumping 0 , length = 2

1) seqArr = {1,1}
i 반복문 [1]. seqArr[i] -> i = length = 2 //seqArr의 2번 인덱스 구하기
 - j 반복문 [1]. j = length = 2
              tmp += seqArr[i-j] = 0 + seqArr[0] = 1
              j -= jumping = 2 - 0 = 2 -> j--
 - j 반복문 [2]. j = 1
              tmp += seqArr [i-j] = 1 + seqArr[1] = 2
              j -= jumping = 1 - 0 = 0 -> j 루프 종료

seqArr[i] = tmp -> seqArr[2] = 2
-> seqArr = { 1, 1, 2 }

2) seqArr = {1, 1, 2}
i 반복문 [1]. seqArr[i] -> i = length = 3 //seqArr의 3번 인덱스 구하기
 - j 반복문 [1]. j = length = 2
              tmp += seqArr[i-j] = 0 + seqArr[1] = 1
              j -= jumping = 2 - 0 = 2 -> j--
 - j 반복문 [2]. j = 1
              tmp += seqArr [i-j] = 1 + seqArr[2] = 3
              j -= jumping = 1 - 0 = 0 -> j 루프 종료

seqArr[i] = tmp -> seqArr[3] = 3
-> seqArr = { 1, 1, 2, 3 }



- 계산 과정 2

initArr -> seqArr = {1, 1, 1, 2}
arr2 = {1,1,1,2}, count 25, bias 1, jumping 0 , length = 4

1) seqArr = {1,1,1,2}
i 반복문 [1]. seqArr[i] -> i = length = 4 //seqArr의 4번 인덱스 구하기
 - j 반복문 [1]. j = length = 4
              tmp += seqArr[i-j] = 0 + seqArr[0] = 1
              j -= jumping = 4 - 0 = 4 -> j--
 - j 반복문 [2]. j = 3
              tmp += seqArr [i-j] = 1 + seqArr[1] = 2
              j -= jumping = 3 - 0 = 3 -> j--
 - j 반복문 [3]. j = 2
              tmp += seqArr [i-j] = 2 + seqArr[2] = 3
              j -= jumping = 2 - 0 = 2 -> j-- -> j 루프 종료 (j > BIAS)

seqArr[i] = tmp -> seqArr[4] = 3
-> seqArr = { 1, 1, 1, 2, 3 }

1) seqArr = {1,1,1,2,3}
i 반복문 [1]. seqArr[i] -> i = 5 //seqArr의 5번 인덱스 구하기
 - j 반복문 [1]. j = length = 4
              tmp += seqArr[i-j] = 0 + seqArr[1] = 1
              j -= jumping = 4 - 0 = 4 -> j--
 - j 반복문 [2]. j = 3
              tmp += seqArr [i-j] = 1 + seqArr[2] = 2
              j -= jumping = 3 - 0 = 3 -> j--
 - j 반복문 [3]. j = 2
              tmp += seqArr [i-j] = 2 + seqArr[3] = 4
              j -= jumping = 2 - 0 = 2 -> j-- -> j 루프 종료 (j > BIAS)

seqArr[i] = tmp -> seqArr[5] = 4
-> seqArr = { 1, 1, 1, 2, 3, 4 }

```
seqArr[4] = 3
seqArr[5] = 4
seqArr[6] = 6
seqArr[7] = 9
seqArr[8] = 13
seqArr[9] = 19
seqArr[10] = 28
seqArr[11] = 41
seqArr[12] = 60
seqArr[13] = 88
seqArr[14] = 129
seqArr[15] = 189
seqArr[16] = 277
seqArr[17] = 406
seqArr[18] = 595
seqArr[19] = 872
seqArr[20] = 1278
seqArr[21] = 1873
seqArr[22] = 2745
seqArr[23] = 4023
seqArr[24] = 5896
25번째 데이터 = 5896
```

- 계산 과정 4.

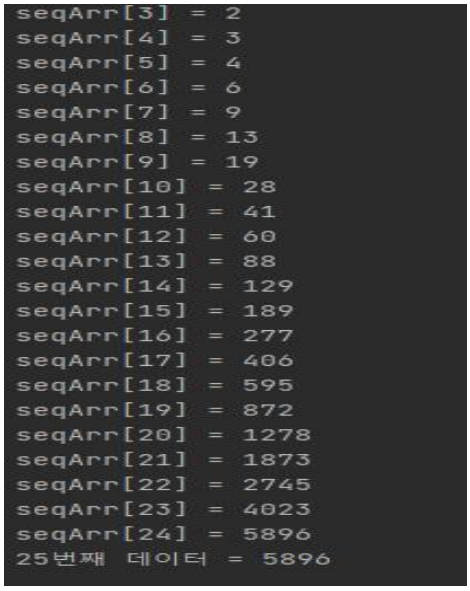
initArr -> seqArr = {1, 1, 1}
arr2 = {1,1,1}, count 25, bias 0, jumping 1 , length = 3

1) seqArr = {1,1,1}
i 반복문 [1]. seqArr[i] -> i = length = 3 //seqArr의 3번 인덱스 구하기
 - j 반복문 [1]. j = length = 3
              tmp += seqArr[i-j] = 0 + seqArr[0] = 1
              j -= jumping = 3 - 1 = 2 -> j--
 - j 반복문 [2]. j = 1
              tmp += seqArr [i-j] = 1 + seqArr[2] = 2
              j -= jumping = 1 - 1 = 0 -> j 루프 종료 (j > BIAS)

seqArr[i] = tmp -> seqArr[3] = 2
-> seqArr = { 1, 1, 1, 2 }

1) seqArr = {1,1,1,2}
i 반복문 [1]. seqArr[i] -> i = 4 //seqArr의 4번 인덱스 구하기
 - j 반복문 [1]. j = length = 3
              tmp += seqArr[i-j] = 0 + seqArr[1] = 1
              j -= jumping = 3 - 1 = 2 -> j--
 - j 반복문 [2]. j = 1
              tmp += seqArr [i-j] = 1 + seqArr[3] = 3
              j -= jumping = 1 - 1 = 0 -> j 루프 종료 (j > BIAS)

seqArr[i] = tmp -> seqArr[3] = 2
-> seqArr = { 1, 1, 1, 2, 3 }



5. 1 ~ 100까지 숫자중 짝수만 출력해보자.
6. 1 ~ 100까지 숫자중 3의 배수만 출력해보자!
7. 1 ~ 100까지 숫자중 4의 배수를 더한 결과를 출력해보자!
8. 1 ~ 100까지 숫자를 순회한다.
   2 ~ 10 사이의 랜덤한 숫자를 선택하고 이 숫자의 배수를 출력해보도록 한다.
9. 1 ~ 100까지의 숫자를 순회한다.
   2 ~ 10 사이의 랜덤한 숫자를 선택하고 이 숫자의 배수를 출력한다.
   다음 루프에서 다시 랜덤 숫자를 선택하고 해당 숫자의 배수를 출력한다.
   그 다음 루프에서 다시 작업을 반복한다.
   끝까지 순회 했을때 출력된 숫자들의 합은 얼마인가 ?
10. 1 ~ 100까지의 숫자를 순회한다.
    9번과 유사하게 2 ~ 10을 가지고 작업을 진행한다.
    다만 이번에는 배수를 찾는게 아니라 랜덤한 숫자가 나온만큼만 이동하고
    이동했을때 나온 숫자들의 합을 계산하도록 만들어보자!

[Bank2Ans5to10Test(main) - SequenceGenerator]

- Bank2Ans5to10Test(main)

```
static final int EVEN = 0;
static final int ODD = 1;

public static void main(String[] args) {
    SequenceGenerator sg = new SequenceGenerator(1, 100);
    sg.createSequence();

    sg.printCondition(EVEN);
    sg.printCondition(ODD);

    System.out.printf("1 ~ 100까지 4의 배수 합: %d\n", sg.findAndSum(4));

    sg.printRandomCondition( randStart: 2,  randEnd: 10);

    // 일단 1 ~ 100을 순회함
    // 2 ~ 10사이의 랜덤값을 매번 변경함
    // 랜덤한 숫자만큼 이동을 함
    sg.printRandomTravel( randStart: 2,  randEnd: 10);

    // 일단 1 ~ 100을 순회함
    // 2 ~ 10사이의 랜덤값을 매번 변경함
    // 랜덤 뽑은 근처리의 배수를 출력해야함
    sg.printRandomTimesTravel( randStart: 2,  randEnd: 10);
}
```

- SequenceGenerator

1. 변수 선언

```
public class SequenceGenerator {
    int start;
    int end;

    int range;

    int[] seqArr;
    int sum;
```

2. 생성자 SequenceGenerator

```
public SequenceGenerator (final int start, final int end) {
    this.start = start;
    this.end = end;

    range = end - start + 1;
    seqArr = new int[range];
}
```

main으로부터 start와 end를 받아서 SequenceGenerator 클래스의
start와 end에 대입한다.
start와 end로 range를 계산한다.
range만큼의 크기를 가지는 seqArr 배열을 생성한다.

3. createSequence

```
public void createSequence () {
    for (int i = start - 1; i < end; i++) {
        seqArr[i] = i + start;
    }
}
```

for문을 이용해 seqArr의 인덱스에 순차적으로 값을 저장한다.
for문을 제어하는 변수인 I는 start -1 = 0부터 I가 end인 100보다 작을 때까지 반복되고,
I + start 값이 각 인덱스에 저장된다.
seqArr[0] = 0 + 1 , seqArr[1] = 1 + 1, seqArr[2] = 2 + 1 , seqArr[3] = 3 + 1 ...
seqArr[99] = 99 + 1

4. printCondition

```java
public void printCondition (int condition) {
    for (int i = start - 1; i < end; i++) {
        if (seqArr[i] % 2 == condition) {
            System.out.printf("seqArr[%d] = %d\n", i, seqArr[i]);
        }
    }
}
```

main으로부터 condition값인 EVEN or ODD를 받아와
for문을 통해 짝수 혹은 홀수만 저장된 인덱스를 출력한다.
- 나머지 연산자의 결과가 EVEN이면 짝수 출력
- 나머지 연산자의 결과가 ODD면 홀수 출력

```
seqArr[75] = 76
seqArr[77] = 78
seqArr[79] = 80
seqArr[81] = 82
seqArr[83] = 84
seqArr[85] = 86
seqArr[87] = 88
seqArr[89] = 90
seqArr[91] = 92
seqArr[93] = 94
seqArr[95] = 96
seqArr[97] = 98
seqArr[99] = 100
```

```
seqArr[80] = 81
seqArr[82] = 83
seqArr[84] = 85
seqArr[86] = 87
seqArr[88] = 89
seqArr[90] = 91
seqArr[92] = 93
seqArr[94] = 95
seqArr[96] = 97
seqArr[98] = 99
```

5. findAndSum

```
public int findAndSum (int find) {
    sum = 0;

    for (int i = start - 1; i < end; i++) {
        if (seqArr[i] % find == 0) {
            sum += seqArr[i];
        }
    }

    return sum;
}
```

main으로부터 전달받은 특정 숫자(find)의 배수를 찾고
배수들의 합을 반환한다.

```
1 ~ 100까지 4의 배수 합: 1300
```

6. printRandomCondition

```
public void printRandomCondition (final int randStart, final int randEnd) {
    int range = randEnd - randStart + 1;

    int rand = (int) (Math.random() * range + randStart);

    for (int i = start - 1; i < end; i++) {
        if (seqArr[i] % rand == 0) {
            System.out.printf("seqArr[%d] = %d\n", i, seqArr[i]);
        }
    }
}
```

main으로부터 전달받은 randStart 값(2)과 randEnd 값(10)을 통해 난수 발생 범위를 계산하고 2~10 사이의 난수를 발생시켜 rand에 저장한다.

for문에서 난수의 배수를 가진 인덱스만 출력한다.

7. printRandomTravel

```
public void printRandomTravel (final int randStart, final int randEnd) {
    int range = randEnd - randStart + 1;

    int rand;

    for (int i = start - 1; i < end; i += rand) {
        rand = (int) (Math.random() * range + randStart);

        System.out.printf("seqArr[%d] = %d\n", i, seqArr[i]);
    }
}
```

main으로부터 전달받은 randStart 값(2)과 randEnd 값(10)을 통해
난수 발생 범위를 계산한다.

for문의 루프마다 난수를 발생시키고, 발생된 난수만큼 이동한 위치의 인덱스를 출력한다.

```
seqArr[76] = 77
seqArr[86] = 87
seqArr[91] = 92
seqArr[96] = 97
seqArr[98] = 99
```

8. printRandomTimesTravel

```
public void printRandomTimesTravel (final int randStart, final int randEnd) {
    int range = randEnd - randStart + 1;

    int rand, tmp = 0;
```

main으로부터 전달받은 randStart 값(2)과 randEnd 값(10)을 통해
난수 발생 범위를 계산한다.
난수를 저장할 변수인 rand를 선언한다.
임시값을 저장할 tmp를 선언하고 0으로 초기화한다.

```java
for (int i = start - 1; i < end; i++) {
    rand = (int) (Math.random() * range + randStart);

    if (seqArr[i] % rand != 0) {
        // 현재 숫자 근처의 rand로 나눠 떨어지는 숫자 찾기
        // 현재 숫자 / rand 의 몫을 보고 나눠 떨어지지 않았으니
        // 여기에 더하기 1 한거에 곱해서 현재값을 뺀 값만큼 + 하면 된다.
        tmp = (seqArr[i] / rand + 1) * rand - seqArr[i];
        i += tmp;
    }

    if (i < 100) {
        System.out.printf("rand = %2d, seqArr[%2d] = %2d\n", rand, i, seqArr[i]);
    }
}
```

for문의 루프마다 난수를 발생시킨다.

I가 100보다 작고, 현재 인덱스 위치의 값이 발생된 난수의 배수라면
발생된 난수인 rand와 rand의 배수에 해당하는 seqArr 배열의 인덱스값을 출력한다.

현재 인덱스 위치의 값이 발생된 난수의 배수가 아니라면
현재 값 근처의 난수의 배수를 찾아야 한다.
현재 값에서 얼마나 이동해야 하는지 계산해 임시값 tmp에 대입한 뒤,
현재 I값에 tmp를 더한다.
위치를 I + tmp만큼 이동시킨 값이 난수의 배수에 해당하는지 검사한다.
ex)
rand = 3
I = 7
현재 위치는 seqArr[7] = 8

tmp =(8 / 3 + 1) * 3 - 8 = 1
I + = tmp -> I = 8

seqArr[8] = 9 이므로 rand(3)의 배수 -> 출력