2018년 Spring 자료구조 실습 과제
#1: Snail Array, Tower of Hanoi

20171665 이 선 호

2018. 04. 04

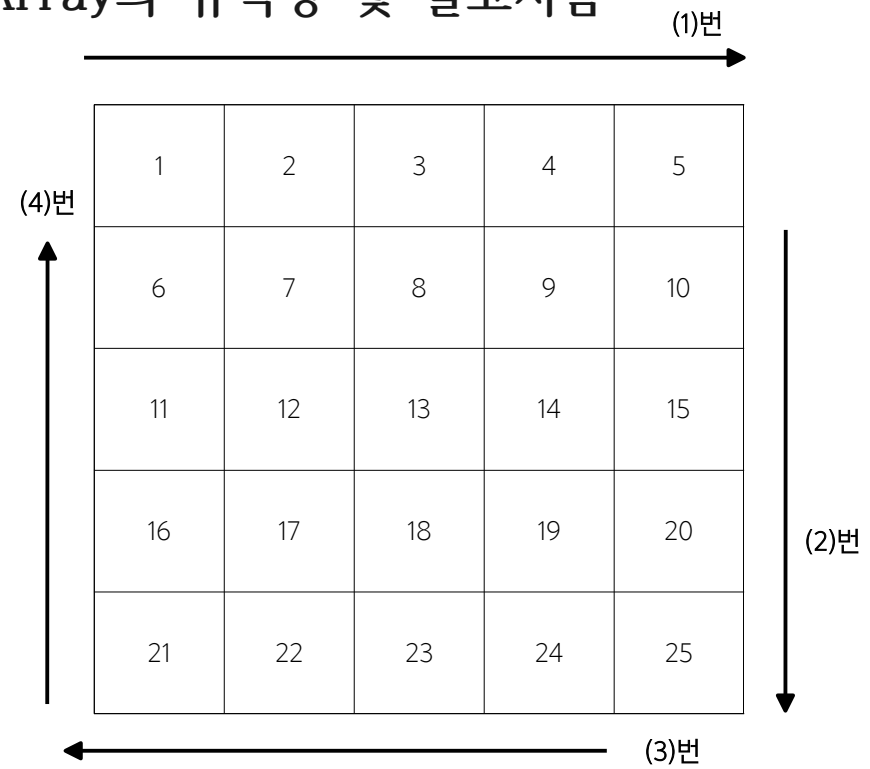
목 차

Ⅰ Snail Array

Ⅱ Tower of Hanoi

I Snail Array

1. Snail Array의 규칙성 및 알고지름



[그림 1] 5 × 5 행렬일 때 Snail Array 채우기

Snail Array의 구현 방식을 자세히 살펴보면 반복되는 규칙성을 찾을 수 있다. 이를 위해 Snail Array의 규칙을 다시 한 번 확인해 보자.

(1) 첫 줄의 가장 왼쪽은 1로 시작하고 오른쪽으로 진해하며 숫자를 증가시킨다.
(2) 그 다음은 아래쪽으로 진행하며 숫자를 증가시킨다.
(3) 그 다음은 왼쪽으로 진행하며 숫자를 증가시킨다.
(4) 그 다음은 다시 위쪽으로 진행하며 숫자를 증가시킨다.
(5) 모든 숫자가 배열될 때까지 이를 반복한다.

(5)번에서 모든 숫자가 배열될 때까지 (1)~(4)번의 과정을 반복한다고 했으므로 (1)번과 (4)번까지의 과정을 loop(반복문)으로 구현할 수 있음을 알 수 있다. 이 loop는 '모든 숫자가 배열될 때까지'가 종결 조건이므로 시작 조건을 필요로 하지 않는 while 문을 채택하는 것이 더 적합하다. 그러면 (1)번에서 (4)번까지의 과정은 while 문 안에 있어야 할 것이다.

또한 (1)~(2)번과 (3)~(4)번에서 규칙성을 발견할 수 있다. [그림 1]에서 (1)번과 (2)번의 과정을 따라가면 'ㄱ'자 형태로 수들을 채워가지만, 반면에 (3)번과 (4)번을 따라가면 'ㄴ'자 형태로 수들을 채워간다. 그러나 수를 채워가는 방향이 반대일 뿐 그 형태는 같다는 것을 알 수 있다. 즉, 채워가는 방향을 나타내는 표시 또는 조건을 추가해주면 (1)~(2)번과 (3)~(4)번 과정은 하나의 같은 주기로 볼 수 있을 것이다.

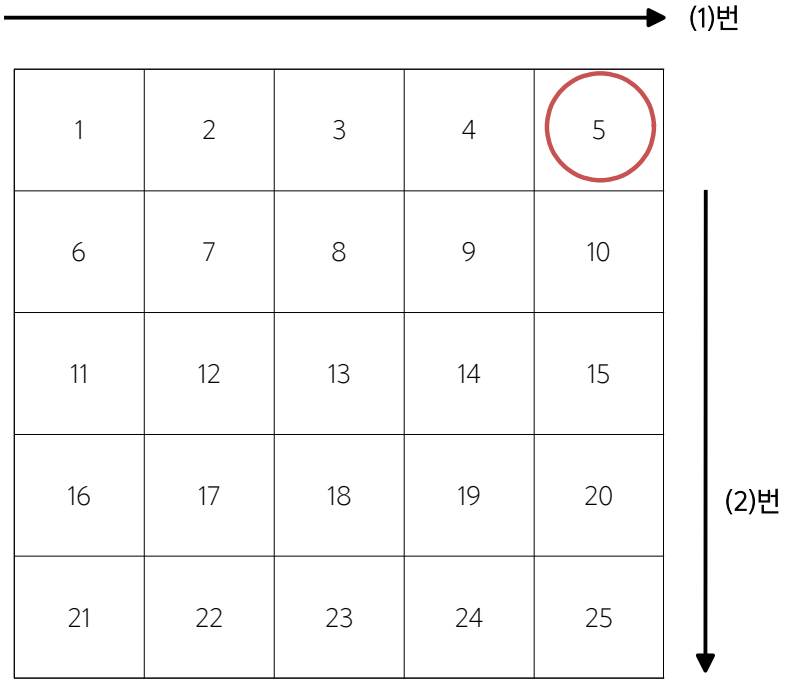
그러면 방향 전환은 어떻게 나타낼 것인가? 이에 대한 답을 찾기 전에 (1)번과 (2)번을 어떻게 구현할 것인지 먼저 생각해 보자. (1)번은 row의 index를 고정해 두고 column의 index 증감에 따라 수를 채워나가고, (2)번은 반대로 column의 index를 고정해두고 row의 index 증감에 따라 수를 채워나갈 것이다. 그러므로 (1)번에서는 수를 채울 수 있는 column의 개수에 유념하여 loop문을 써야 하고, (2)번에서는 row의 개수에 유념해야 한다. 이를 의사코드(pseudocode)로 나타내면 다음과 같다.

```
for(int i = 0; i < row를 고정해 두고 채워나갈 수 있는 column의 개수; i++){
    (1)번 과정
}
for(int i = 0; i < column을 고정해 두고 채워나갈 수 있는 row의 개수; i++){
    (2)번 과정
}
```

(1)번의 for문에서는 종결되기 전까지 column의 index를 증가시키며 수를 채워나갈 것이다. 그러므로 index는 1씩 증가해야 한다. 마찬가지로 (2)번의 for문에서는 row의 index를 1씩 증가시킬 것이다. 여기서 (1)~(2)번과 (3)~(4)번의 방향 전환에 관한 힌트를 얻을 수 있다. (1)~(2)번에서는 row 기준으로 오른쪽 방향 또는 column 기준으로 아래 방향으로 수를 채워나가므로 index를 1씩 증가시켜주지만, 반면에 (3)~(4)번에서는 왼쪽 방향 또는 위 방향으로 수를 채워나가므로 index를 1씩 감소시켜준다. 그래서 row 또는 column의 index를 증가시키는 변수(실제 코드에서는 'way'라고 할 것이다)를 하나 선언해서 (1)~(2)번 과정을 수행할 때는 1로, (3)~(4)번 과정일 때는 -1로 초기화해주면 될 것이다. 그런데 1에서 -1 또는 -1에서 1로 변경할 때는 처음 값에 -1을 곱해주면 되므로 (1)~(2)번 과정을 마무리하는 구간에서 방향 전환 변수에 -1을 곱해주는 과정을 추가하면 된다.

```
for(int i = 0; i < row를 고정해 두고 채워나갈 수 있는 column의 개수; i++){
    (1)번 과정
}
for(int i = 0; i < column을 고정해 두고 채워나갈 수 있는 row의 개수; i++){
    (2)번 과정
}
way *= -1;
```

그러면 결국 index를 증감시키기 위해서 for문 안의 index에 변수 way의 값을 더해주는 과정을 넣으면 될 것이다. 그런데 이 과정은 어느 위치에 넣는 것이 맞을까? 배열의 각 항에 수를 넣어주고 index를 증감시키는 것이 좋을까, 아니면 index를 먼저 증감시키는 것이 좋을까? [그림 2]를 보면서 자세히 살펴보자.



[그림 2] 한 줄을 다 채우고 나면 다음 한 줄의 한 entry를 차지하게 되는 현상

(1)번 과정을 시행하고 (2)번 과정을 시행하는 순간, [그림 2]에서 볼 수 있다시피 (2)번 과정을 시행할 column의 한 entry를 이미 사용하게 된다. 그렇기 때문에 column 4에 수를 채워가기 직전에 row의 index에 1을 더해주는 과정이 필요하다. 마찬가지로 (2)번에서 (3)번 과정으로 넘어갈 때도 column의 index에서 1을 빼주는 과정이 필요하다. 이와 같이 해야 이미 수를 넣은 자리를 또 한 번 침범하거나 index가 배열 밖을 벗어나는 일을 막을 수 있다. 그러므로 index에 1을 더하거나 빼주는 과정을 for문 안의 앞부분에서 시행하고 그 다음에 수를 대입하는 과정을 거치는 것이 합리적이다. 대신, 처음에 (1)번 과정을 시행할 때 column의 index를 -1에서 시행해야 할 것이다. 왜냐하면 column의 index에 1을 더하는 과정을 먼저 시행하므로 만일 0부터 시작할 경우 for문을 종결하고 나면 column의 index가 5가 되어 배열의 가장자리 밖으로 넘어가기 때문이다.

마지막으로 생각해야 할 것은 각 과정마다 채워나가야 하는 column 또는 row의 개수일 것이다. 하지만 이는 앞에서 서술했던 것을 다시 적용하면 쉽게 답을 얻을 수 있다. 처음 (1)번 과정을 시행하고 나면 다음 (2)번 과정에서 넣어야 할 entry의 개수가 한

개 줄어듦을 앞에서 언급한 바가 있다. 그러므로 (1)번 과정을 수행하고 나면 채워나가야 하는 row의 수를 1 감소시키고, (2)번 과정을 수행하고 나면 column의 수를 1 감소시키면 될 것이다. 그리고 채워나가야 하는 row 또는 column의 수가 0이 되는 순간 우리는 while 문을 종결시키면 된다.

지금까지의 규칙성을 바탕으로 코드를 작성하면 다음과 같다.

```
void function SnailArray
    while (1) {
        for (int i = 0; i < col; i++, number++) {
            num2 += way;
            arr[num1][num2] = number;
        }
        row--; // (1)번 과정
        if (row < 0 || col < 0) break; // 종결 조건
        for (int i = 0; i < row; i++, number++) {
            num1 += way;
            arr[num1][num2] = number;
        }
        col--; // (2)번 과정
        way *= -1; // 방향 전환
    }
```

2. 사용한 자료구조

Snail Array을 나타내기 위해 이차원 배열을 선언해야 할 것이다. 그런데 단순히 이차원 배열을 선언하는 것만으로는 무리가 있다. 그 이유는 다음과 같다.

- Input으로 받고자 하는 n의 최댓값이 정해져 있지 않다. MAX_SIZE 크기의 행과 열의 배열을 선언하면 공간 낭비가 크다.
- 일반적으로 arr[n][n]의 선언은 적절하지 않다.

그래서 정적 할당보다는 동적 할당으로 배열을 선언하는 것이 더 합리적이다. 그래서 더블 포인터로 변수명을 선언하고, malloc을 이용해 n개의 행과 열을 가진 배열의 메모리 공간을 할당한다.

```
int **arr;
arr = (int**)malloc(sizeof(int*)*n);
for (int i = 0; i < n; i++) {
	arr[i] = (int*)malloc(sizeof(int**)*n);
}
```

3. 실제 구현 결과

[그림 3] N값이 7일 때 Snail Array가 출력된 결과물

※ 실제 코드는 압축파일에 첨부되어 있음.

Ⅱ Tower of Hanoi

1. Tower of Hanoi의 일반화

세 기둥 A, B, C가 있고 크기가 다른 3개의 원반들이 기둥 A에 꽂혀 있다고 가정한다. 그리고 이 원반들은 크기가 작은 것부터 위에서 아래로 차례대로 꽂아 있어야 한다는 규칙을 따른다. 이처럼 기둥 A에 꽂혀 있는 원반들을 규칙을 지키면서 기둥 B로 모두 옮겨야 한다면 어떠한 알고리즘을 따라야 할까?

크기가 작은 것부터 큰 것 순으로 원반 1, 2, 3으로 번호를 붙인다. 우선 기둥 A의 가장 위에 꽂혀 있는 원반 1을 기둥 A에서 B로 옮겨야 한다. 그리고 원반 2를 기둥 A에서 C로 옮기고, 기둥 B에 있는 원반 1을 기둥 C로 옮긴다. 그러면 기둥 B는 아무것도 꽂혀 있지 않을 것이고 기둥 A의 맨 밑에 있었던 원반 3을 기둥 B로 이동시킬 수 있다. 그 이후 나머지 두 개의 원반 1과 2도 기둥 C에서 B로 옮겨야 한다. 기둥 C 맨 위에 있는 원반 1은 기둥 A로 이동시키고, 그 다음에 원반 2는 기둥 B로 이동시킨다. 마지막으로 기둥 A에 있는 원반 1을 기둥 B로 이동시킨다.

이와 같은 과정을 살펴보면 다음과 같은 결론을 도출해낼 수 있다.

▶ 원반 1, 2, 3을 기둥 A에서 B로 옮긴다고 하자.
(1) 원반 3을 제외한 나머지 원반들을 기둥 A에서 C로 옮긴다.
(2) 원반 3을 기둥 A에서 B로 옮긴다.
(3) 원반 3을 제외한 나머지 원반들을 기둥 C에서 B로 옮긴다.

▷ (1)에서 원반 1과 2를 기둥 A에서 C로 옮기는 과정도 위의 과정과 닮아있다.
(1) 원반 2가 아닌 원반 1을 기둥 A에서 B로 옮긴다.
(2) 원반 2를 기둥 A에서 C로 옮긴다.
(3) 원반 2가 아닌 원반 1을 기둥 B에서 C로 옮긴다.

▷ (3)에서 원반 1과 2를 기둥 C에서 B로 옮기는 과정도 위의 과정과 닮아있다.
(1) 원반 2가 아닌 원반 1을 기둥 C에서 A로 옮긴다.
(2) 원반 2를 기둥 C에서 B로 옮긴다.
(3) 원반 2가 아닌 원반 1을 기둥 A에서 B로 옮긴다.

▶ 마찬가지로 크기가 다른 원반 4개를 기둥 A에서 B로 옮긴다고 한다면
(1) 원반 4를 제외한 나머지 원반들을 기둥 A에서 C로 옮긴다.
(2) 원반 4를 기둥 A에서 B로 옮긴다.
(3) 원반 4를 제외한 나머지 원반들을 기둥 C에서 B로 옮긴다.

이를 통해 우리는 원반을 옮기는 과정을 일반화할 수 있다.

▶ 원반 1, 2, … , $n-1$, n을 기둥 A에서 B로 옮긴다고 하자.
(1) 원반 n을 제외한 나머지 원반들을 기둥 A에서 C로 옮긴다.
(2) 원반 n을 기둥 A에서 B로 옮긴다.
(3) 원반 n을 제외한 나머지 원반들을 기둥 C에서 B로 옮긴다.
※ 단, n이 1일 때는 원반 1을 기둥 A에서 B로 옮기는 과정만 한다.

2. 사용한 자료구조 및 알고리즘

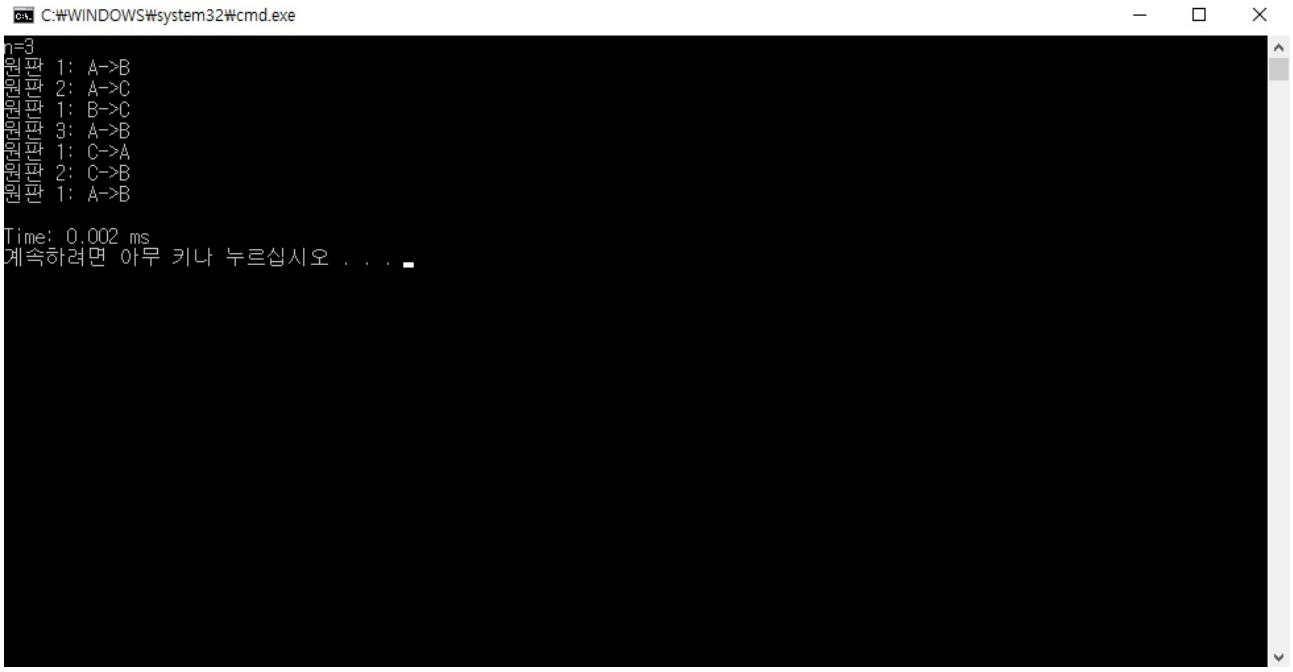
1에서 원반을 한 기둥에서 다른 기둥으로 옮기는 과정을 일반화할 수 있었으므로 재귀함수를 써서 코드를 작성할 수 있다.

```
void move (int n, char from, char to){
        // 원반 n을 기둥 from에서 to로 옮긴다.
        return;
}

void hanoi(int n, char from, char by, char to){
        if(n == 1) move(n, from, to);
        else{
                hanoi (n - 1, from, to, by); // 원반 n을 제외한 나머지 원반들을 기둥 from에서 by로 옮긴다.
                move (n, from, to); // 원반 n을 기둥 from에서 to로 옮긴다.
                hanoi (n - 1, by, from, to); // 원반 n을 제외한 나머지 원반들을 기둥 by에서 to로 옮긴다.
        }
        return;
}
```

여기서 from은 현재 원반(들)이 위치해 있는 기둥이고 to는 원반들을 옮겨 최종적으로 위치하게 하고자 하는 기둥의 이름을 의미한다. by는 원반들을 기둥 to로 옮기기 위해 임시로 거치는 기둥의 이름을 뜻한다.

3. 실제 구현 결과



[그림 4] n이 3일 때 출력된 결과

※ 실제 코드는 압축파일에 첨부되어 있음.

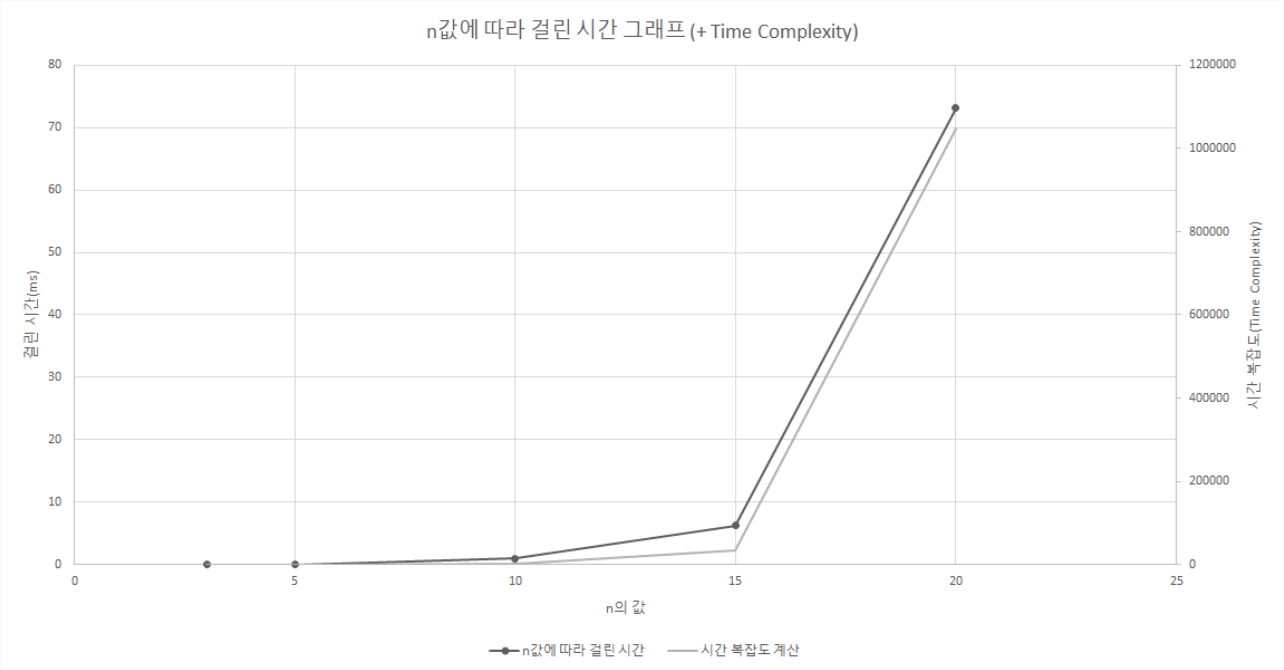
4. 예시에 따라 걸린 시간과 분석

n값이 각각 3, 5, 10, 15, 20일 때 걸리는 시간을 구하였다.

n의 값	3	5	10	15	20
걸린 시간(ms)	0.001	0.002	0.904	6.194	78.161

[표 1] n값에 따라 걸린 시간

이 데이터로부터 그래프를 그리면 다음과 같다.



[그림 5] n값에 따라 걸린 시간과 시간 복잡도 그래프

[그림 5]에 의하면 n값에 따라 걸린 시간 그래프는 시간 복잡도 2^n 그래프와 거의 비슷한 모습을 보이고 있다. 즉, 이 프로그램의 시간 복잡도는 $O(2^n)$임을 알 수 있다. 이를 증명하는 과정은 다음과 같다.

(1) 원반 n개를 A에서 B로 옮기는 과정은 원반 n−1개를 A에서 C로 옮기고 가장 큰 원반 n을 A에서 B로 옮긴 다음에 원반 n−1개를 C에서 B로 옮기는 것임을 알고 있다. 함수를 한 번 실행하는 것을 1로 생각하고 원반 n개를 옮기는 과정의 시간 복잡도를 a_n이라고 하면 아래와 같은 식을 도출할 수 있다.

$$a_n = 2a_{n-1} + 1$$

(2) 위 식을 풀이하여 일반항을 구하면 다음과 같다.

$$a_1 = 1$$

$$a_n + 1 = 2a_{n-1} + 2$$

$$a_n + 1 = 2(a_{n-1} + 1)$$

$a_n + 1 = b_n$이라고 하면 $b_n = 2b_{n-1}$, 그리고 $b_1 = 2$ $(\because a_1 + 1 = 2)$이므로 b_n은 2^n이다.

따라서 $a_n = 2^n - 1$이다.

(3) a_n이 $2^n - 1$이면 $big O$는 $O(2^n)$이다.