Algorithm

- 1주차-

제출일 : 2020 / 09 / 07
분반 : 01
학번 : 201702068
이름 : 전병민

1. 과제 1

- 문제 설명 : N개의 정수가 주어져 있을 때, 최대값과, 최소값을 구하는 문제이다.

- 해결 방법 : 효율적인 프로그램 구현을 위해 **Divide and Conquer 알고리즘을 사용하였다.**

- 코드 설명 :

1. 전역변수

```
public static int max = Integer.MIN_VALUE;
public static int min = Integer.MAX_VALUE;
public static int arr[];
```

(min, max, arr[]는 전역변수로 선언하였고, min의 초기값은 int의 최대값, max의 초기값은 int의 최소값을 넣어주었다.)

2. findMaxMin()

```
public static void findMaxMin(int left, int right) {
    if(left == right) {
        max = arr[left];
        min = arr[left];
    } else {
        int mid = (left+right)/2;
        findMaxMin(left, mid);

        int max1 = max;
        int min1 = min;

        findMaxMin( left: mid+1, right);

        if(max < max1){
            max = max1;
        }
        if(min > min1){
            min = min1;
        }
    }
}
```

findMaxMin()의 전체 코드이다.

```
if(left == right) {
    max = arr[left];
    min = arr[left];
```

먼저 arr[]가 가장 작은 부분 문제로 나누졌을 때이다.
이때의 max와 min의 값은 arr[left](= arr[right])이다.

```
} else {
    int mid = (left+right)/2;
    findMaxMin(left, mid);

    int max1 = max;
    int min1 = min;

    findMaxMin( left: mid+1, right);

    if(max < max1){
        max = max1;
    }
    if(min > min1){
        min = min1;
    }
}
```

그다음 arr[]가 가장 작은 부분문제로 나뉘지 않았을 때에는 서브 리스트의 크기가 1일 될 때까지 문제를 왼쪽과 오른쪽 두 부분으로 recursive하게 나눠주었다.

arr[]를 비슷한 크기로 나누기 위해서 left와 right의 중간값인 mid를 만들어주었다.
mid를 기준으로 왼쪽의 부분문제의 최대값을 max1로, 최소값을 min1로 하였다.

오른쪽의 부분문제의 최대값은 max에 재할당, 최소값은 min에 재할당하였다.
(교수님의 강의 내용과 비교하자면, 재할당된 max가 max2의 역할, min이 min2의 역할을 한다.)

max와 max1을 비교하여 max가 더 작다면 max에 max1값을 재할당해준다.
마찬가지로 min과 min1을 비교하여 min이 더 크다면 min에 min1값을 재할당해준다.

이렇게 recursive하게 전체의 max와 min을 찾아낼 수 있다.

3. main()

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);
    int n = sc.nextInt();
    arr = new int[n];

    for (int i = 0; i < n; i++){
        arr[i] = sc.nextInt();
    }

    findMaxMin( left: 0,  right: arr.length-1);

    System.out.println("Min : " + min);
    System.out.println("Max : " + max);
}
```

for문으로 arr[]에 값을 입력받았고 findMaxMin()을 이용해 전역변수 min, max에 값을 넣었다.

➢ 결론 :

N개의 정수가 주어졌을 때, 최대값과 최소값을 찾는 문제에서 **Divide and Conquer**를 이용한다면, 2n-3 비교보다 적은 비교를 하기 때문에 더 효율적인 프로그램을 만들 수 있다.

2. 과제 2

- 문제 설명 : 프로세스 쌍이 주어져 있을 때 교착상태인 프로세스들을 구하는 문제이다.
- 해결 방법 : 그래프 안에서 사이클(교착상태)을 구하기 위해서 **DFS를 이용하였다.**
- 코드 설명 :

1. 전역변수

```
public static boolean[][] edge;
public static boolean[] visit;
public static boolean isDeadlock;
public static int firstId;
public static int numberOfProcesses;

public static List<Integer> firstProcessId = new ArrayList<>();
public static List<Integer> secondProcessId = new ArrayList<>();
public static List<Integer> deadlockProcess = new ArrayList<>();
```

edge[][] : findDeadlock() 함수에서 프로세스가 작업을 기다리는 중인지를 판단한다.

visit[] : findDeadlock() 함수에서 해당 프로세스를 들렀는지 판단한다.

isDeadlock : 어떤 프로세스가 주어졌을 때 그 프로세스가 교착상태인지 판단한다.

firstId : 어떤 프로세스가 주어졌을 때 그 프로세스를 담는 변수이다.

numberOfProcesses : 정수 쌍의 개수를 입력받기 위한 변수이다.

ArrayList 3개 : 교착상태인 프로세스들의 출력을 위한 리스트이다.

2. main()

```
public static void main(String[] args) {
    Scanner sc = new Scanner(System.in);

    System.out.print("프로세스 정수 쌍을 총 몇개 입력할 것인지 입력해주세요 : ");
    numberOfProcesses = sc.nextInt();
    System.out.println("공백을 이용하여 프로세스 정수 쌍을 입력해주세요.");
    visit = new boolean[101];
    edge = new boolean[101][101];
    int num1, num2;
    for (int i = 1; i <= numberOfProcesses; i++){
        num1 = sc.nextInt();
        num2 = sc.nextInt();
        edge[num1][num2] = true;
    }
```

먼저 main()의 입력부분이다.

입력의 편의를 위해 프로세스 정수 쌍을 몇 개 입력받을 것인지에 대한 sc.nextInt()를 추가하였다.

배열의 크기는 문제에 범위가 따로 주어지지 않았기 때문에 예시의 데이터 값만은 받을 수 있도록 설정하였다.

edge[][] 인덱스를 이용하여 (x, y)형태의 정수 쌍을 입력받고 true로 처리하였다. (true면 x가 y가 작업이 끝나길 기다리고 있는 것)

visit[] 의 초기 값은 모두 false여야 하기 때문에 main()에서 따로 값을 넣어주지 않았다.

```
    for (int j = 1; j <= 100; j++){
        firstId = j;
        findDeadlock(j);
        if (isDeadlock){
            removeExtra();
            System.out.print(j+"와 관련된 프로세스 체인 : ");
            for (int i = 0; i < firstProcessId.size(); i++){
                System.out.print("(" + firstProcessId.get(i) + " " + secondProcessId.get(i) + ")");
                if (secondProcessId.get(i) == firstId && i < firstProcessId.size()-1){
                    System.out.print(" | ");
                }
            }
            deadlockProcess.add(firstProcessId.get(0));
            System.out.println();
        }
        secondProcessId.clear();
        firstProcessId.clear();
        for (int i = 1; i <= 100; i++){
            visit[i] = false;
        }
        isDeadlock = false;
    }
    System.out.println();
    System.out.println("[교착상태인 프로세스]");
    System.out.println(deadlockProcess);
}
```

main()에서 findDeadlock(), removeExtra() 호출 및 출력에 관련된 부분이다.

반복문으로 어떤 프로세스ID를 findDeadlock()에 넣었고 findDeadlock()에서 isDeadlock이 true가 된다면 교착상태인것이기 때문에 교착상태가 아니라면 출력도 하지 않고 다음 프로세스ID로 넘어가는 형식이다.

isDeadlock이 어떤 값이던지, 다음 프로세스ID가 교착상태인지 판단하기 위해서 출력를 위한 리스트와 visit[], isDeadlock를 모두 초기화 시켜주었다.

deadlockProcess는 교착상태인 프로세스들을 모두 모아주기 위해서 isDeadlock일때 반복문이 돌아갈 때 해당 프로세스를 리스트에 add해주었다.

3. findDeadlock()

```
public static void findDeadlock(int i){

    visit[i] = true;

    for (int j = 1; j < 100; j++){
        if (edge[i][j] && !visit[j]) {
            firstProcessId.add(i);
            secondProcessId.add(j);
            findDeadlock(j);
        } else if (edge[i][j] && visit[j] && firstId == j){
            firstProcessId.add(i);
            secondProcessId.add(j);
            isDeadlock = true;
        }
    }
}
```

findDeadlock()함수는 교착상태를 찾아내는 함수이다.
먼저 i 라는 processID에 방문한 것이기 때문에 visit[]를 true로 해준다.

edge[i][j]에서 j에 해당하는 부분을 일일이 찾아내기 위해 반복문을 이용하였다. edge[i][j]가 true라는 뜻은 i와 j가 연결되어 있다는 것이고, 이 문제에서는 i가 j의 작업이 끝나길 기다리고 있다는 뜻이기 때문에 true 일 때라는 조건이 들어가 있다.

첫번째 if에서 visit[]이 false일 때에는 아직 해당 프로세스를 방문하지 않았기 때문에 출력리스트에 첫번째 값과 두번째 값을 저장해주었다. 여기서 findDeadlock(j)를 재귀적으로 돌아주는데, 이 함수를 돌면서 출력리스트에 쓸 모 없는 값까지 같이 add되기 때문에 removeExtra()라는 함수로 잘 못 들어간 프로세스들을 전부 지우는 작업을 해주었다. (이 재귀함수는 반복문이 다 돌아갔을 때 멈추게 된다.)

재귀함수가 돌면서 만약 firstID와 j가 같다면 다시 처음 프로세스로 되돌아 온 것이기 때문에 이것을 교착상태라고 하였다. isDeadlock를 true로 바꿔주고 출력리스트에 값을 넣었다.

4. removeExtra()

```
public static void removeExtra(){
    for (int i = 1; i < firstProcessId.size(); i++){
        if (firstProcessId.get(i) != secondProcessId.get(i-1)){
            if (secondProcessId.get(i-1) == firstId){
                break;
            }
            firstProcessId.remove( index: i-1);
            secondProcessId.remove( index: i-1);
            i = 1;
        }
    }
    int last = firstProcessId.size()-1;
    while(secondProcessId.get(last) != firstId){
        firstProcessId.remove(last);
        secondProcessId.remove(last);
        last = firstProcessId.size()-1;
    }
}
```

removeExtra()는 findDeadlock에서 얻은 출력리스트에서 쓸모없는 값을 지워지기 위해 만들었다. 이 문제에서 핵심 함수는 아니지만 설명하자면

첫번째 반복문은 출력리스트를 처음부터 돌면서 쓸모없는 값들을 제거해주었고
두번째 반복문은 출력리스트를 뒤에서 돌면서 마지막 값이 첫번째 프로세스가 나올 때까지 돌아주며 값을 제거해주었다.

➢ 결론 :
교착상태를 구하는 문제 또한 그래프를 응용하여 구할 수 있으며 또한 그래프가 주어진 상태에서 **DFS알고리즘을** 사용하여 사이클을 얻어낼 수 있다.