알고리즘분석

**HW#2 Report**

2011147068 김정환

2011147115 허재화

2012147562 최인호

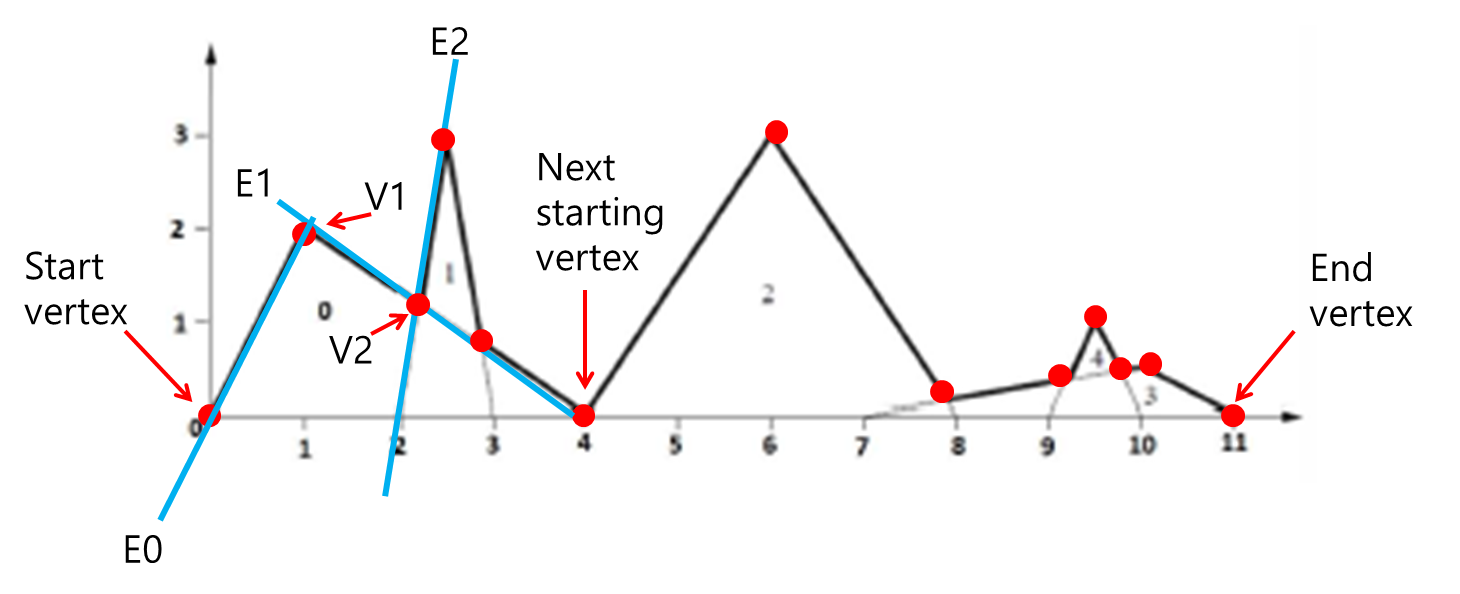
1. **목차**
   1. **문제 분석**
   2. **알고리즘**
   3. **구현**
      1. **Class 설명**
      2. **알고리즘에 대한 세부구현**
   4. **토의 과정**
      1. **Big-O notation을 통한 분석**
      2. **구현 과정에서 겪은 어려움**

**문제 분석**

* Input.txt의 구성은 전체 case number로 이루어 진다. 각 case는 들어오는 총 building의 개수와 left, top, right의 좌표가 순서대로 들어온다.
  + 각 test case에 대해, input인 Vertex 및 Edge의 개수는 각각 2 × (building\_num,) 3× (building\_num)이다.
  + 한 vertex에 대해 여러 building이 공유할 수 있다.
  + Input으로부터 입력 받는 Building의 위치는 어떠한 순서도 없이, 독립적으로 들어온다.
* 이 때 각 점의 x 좌표의 크기에 대해 오름차순을 하면, x\_left <= x\_top <= x\_right을 만족한다.
  + Building의 형태는 acute triangle 혹은 right triangle이 될 수 있다.
* Output은 building들을 정면에서 빛을 투과 할 때 하늘과 맞닿는 점들의 좌표의 집합이다.
  + 다른 building에 의해 가려지는 Vertex들이 있고, building들 끼리 만나서 생성되는 Vertex들이 있다.

1. **알고리즘**

* 지상과 맞닿아 있는 점들 중 가장 왼쪽에 있는 Vertex와 가장 오른쪽에 있는 Vertex를 두 점을 각각 시작점과 끝점으로 한다..
* 찾은 Vertex부터 Traverse를 시작하는데 그 과정은 다음과 같다.
  + 현재 선택한 점을 포함하는 Edge 중에서 가장 적절한 Edge를 선택한다.
  + 선택한 Edge에서 현재 점과 가장 가까운 위치의 점을 다음 점으로 선택한다,
  + 이 과정을 ( x, 0 )에 도달할 때 까지 수행한다.
* 도달한 ( x, 0 )이 끝점과 같은 점인지 비교한다.
  + 끝점과 같다면, Traverse를 종료한다,
  + 끝점과 다르다면, 도달한 ( x , 0 )의 x 값보다 크거나 같은 다음 시작점을 선택하고, Traverse 과정을 다시 시작한다.
* 예제로 주어진 그림을 가지고 하면 아래와 같다,

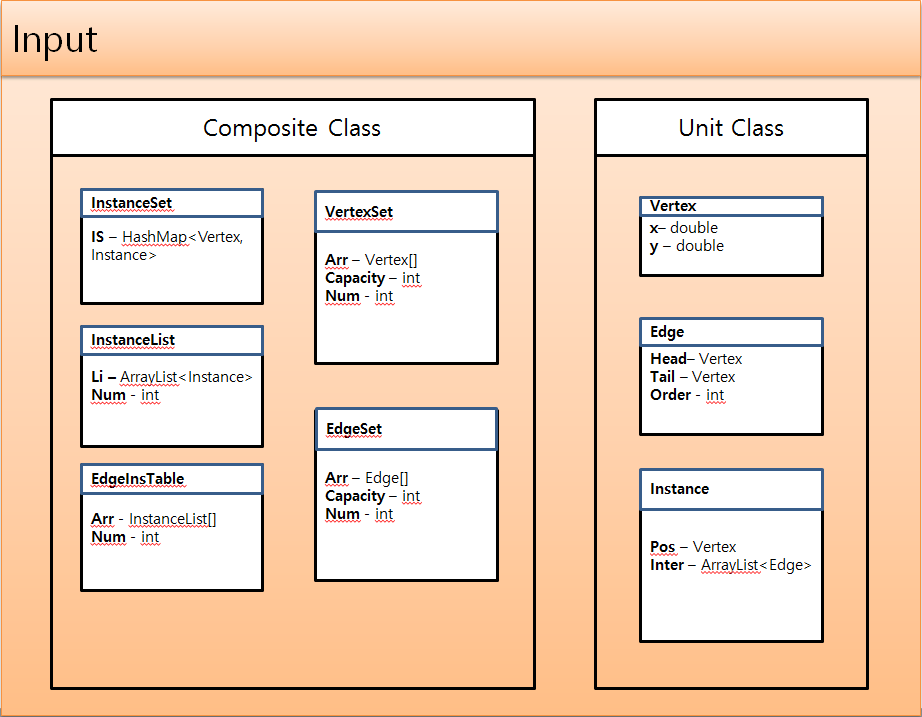


* + ( 0, 0 )와 (11,0)이 각각 Start vertex와 End vertex가 된다.
  + Start vertex부터 Traverse를 시작한다.
    - Start vertex를 지나는 것은 E0밖에 없으므로 E0이 선택된다,
    - E0을 지나는 Vertex는 V1밖에 없으므로 다음 vertex는 V1이 선택된다.
  + 다음 Vertex는 V1.,
    - V1을 지나는 Edge는 E0와 E1이 있는데, 둘 중 더 적절한 E1을 선택한다
    - E1위에는 V1, V2, … 등 여러 Vertex들이 있는데 이중 가장 적절한 V2를 선택한다.
  + 이 과정을 반복하다 (4, 0) 에 도달했을 때, (4, 0) 이 End vertex인지를 비교한다. 현재 (4 , 0)은 End vertex가 아니므로, Program은 새로운 시작점을 찾는다.
  + Next starting point로 (4,0)이 선택되고 Traverse를 다시 반복한다.
  + 결국 End Vertex에 도달하고 Traverse는 종료된다.
* 한 Vertex에서 가장 적절한 Edge를 선택하는 과정은 다음과 같다.
  + 현재 vertex를 지나는 모든 Edge들은 크게 Right edge와 Non-right edge로 나뉜다.
  + Right Edge중에서 가장 적절한 Edge는 y값이 가장 큰 것이다.. 정면에서 봤을 때 y값이 작은 다른 edge들이 가려지기 때문이다.
  + Non-right edge중 가장 적절한 Edge는 기울기가 가장 큰 Edge이다. Right Edge인 경우와 같은 이유이다.
  + 만약 Proper right edge와 Proper non-right edge를 모두 가지고 있는 경우, 이전에 선택한 Edge가 Right인지 아닌지를 통해 Final proper edge를 선택한다.
* 선택한 Edge에서 가장 적절한 Vertex를 구하는 과정은 다음과 같다.
  + 현재 점과 선택한 Edge 위의 점들 중에서 가장 가까운 점을 구하되, Traverse하는 방향을 고려하여 선택한다.
    - 밑으로 Traverse 하는 경우, 현재 Vertex보다 아래에 있되, 가장 가까운 Edge 상의 Vertex를 구한다.
    - 위로 Traverse 하는 경우, 현재 Vertex보다 크되, 가장 가까운 Edge 상의 Vertex를 구한다..

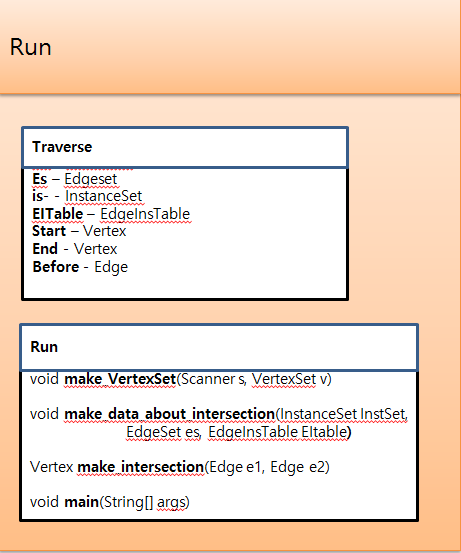
1. **구현**

**2.1 Class 설명**

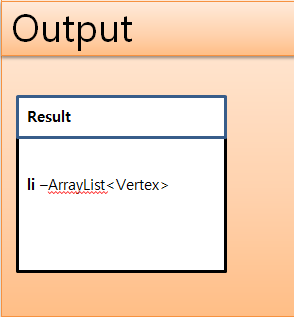
* 프로그램은 Input, Run, Output, 총 3개의 Package로 구성된다.
* **Input** : 알고리즘을 구현할 때 사용되는 객체들로 구성된다.



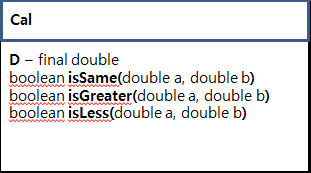
* + **Composite Class**
    - InstanceSet : Vertex를 통해 Instance를 찾기 위해 만든 객체
    - InstanceList : Instance로 만들어진 List의 Wrapper 객체
    - EdgeInsTable : Edge의 들어온 순서를 index로 하여 해당 edge에 대한 InstanceList를 Table로 만든 객체.
    - VertexSet : Input.txt로부터 받는 Vertex들의 Set. 좌표값이 같은 Vertex는 먼저 들어온 Vertex만을 저장하고, 나중에 들어온 Vertex 자리에는 먼저 들어온 Vertex를 넣는다.
    - EdgeSet : Input.txt로부터 들어온 순서대로 Edge를 저장하는 객체.
  + **Unit Class**
    - Vertex : 각 점들을 나타내는 객체
    - Edge : 각 선분들을 나타내는 객체, 들어오는 순서를 order에 함께 저장하는데, order는 Edge\_Instance Table에서 index로 사용하기 위함이다.
    - Instance : 어떤 Vertex 및 그 Vertex를 지나는 Edge들을 나타내는 객체
* **Run**



* + Traverse : Input의 클래스들(EdgeSet, InstanceSet 등등..)을 데이터로 하여, Traverse를 수행하는 클래스. Before라는 값을 두어서, 한 스카이 라인에 여러 개의 점이 나오는 경우를 구분했다.
  + Run
    - void **make\_VertexSet**(Scanner s, VertexSet v) : Input.txt로부터 값을 읽으며 바로 Vertex set에 저장하기 위한 method
    - void **make\_data\_about\_intersection**(InstanceSet InstSet, EdgeSet es, EdgeInsTable EItable**) :** Edgeset의 edge들을 서로 비교를 하며 교점을 찾고, 그 교점에 대한 Instance를 만드는 것 및 EItable에 값을 갱신하는 method
    - Vertex **make\_intersection**(Edge e1, Edge e2) : 두 edge에 대해 교점이 만들어지면 만들어진 교점을 return 하고 아니면 null을 return 한다.
    - void **main**(String[] args) : Input을 받아들이는 과정 및 Traverse 모든 과정이 수행된다.
* **Output**



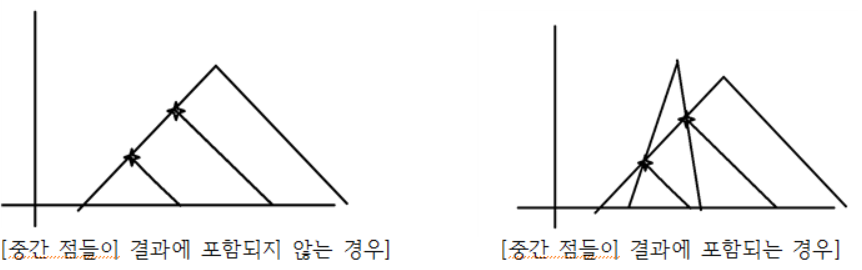
* Result : Traverse의 결과를 담기 위한 객체. Skyline을 traverse할 때 왼쪽에서 오른쪽으로 가기 때문에 Output 형식에 맞는 출력이 자동으로 된다.
* **기타**



* 실수형의 값을 비교하기 위한 객체
* 실수형끼리 ==, >, < 으로 비교할 때는 매우 작은 차이 조차도 용인이 안된다. 그래서 0.01(아마도 D값에 따른)정도의 정밀도로 비교하기 위해 만들었다.

**2.2 알고리즘에 대한** 세**부 구현**

* **교점찾기**
* 처음 3 × linenum의 vertex들을 입력 받고, 그 vertex들로부터 각 삼각형마다 2개의 edge(left\_top, top\_right)들을 만들어서 저장한다. 그리고 모든 edge들로부터 다른 edge와 만나서 생기는 교점들을 찾는다. 그 과정은 다음과 같다.
  + 모든 edge들에 대해서 자신이 아닌 다른 모든 edge들과 교점이 생기는지 확인해야 한다. 여기서 교점의 개수는 n 이므로 총 의 경우에 대해서 교점 발생 여부를 조사해야 한다. 그리고 여기에서 경우를 나눈 게, edge가 평행하거나 겹치는 경우이다. 그럴 때는 단순히 기울기가 다른 두 edge가 교점을 발생시키는 것과는 다르게 처리를 해줘야 한다.
    - 평행하고 겹치지 않는 경우(교점이 발생하지 않는 경우)
      * 두 edge의 기울기가 같은 경우에 대하여, 두 edge의 시작점과 끝점을 이용해 직선의 방정식을 구한다음 x값이 0일 때 방정식의 값을 구해서 그 값을 비교한다. 그 값이 다르면 그 edge들은 교점을 발생시키지 않는 것이다. 그럴 때는 null을 리턴해 주면서 make\_intersection() 함수를 종료해준다.
    - 평행하고 겹치는 경우 (교점이 무수히 많은 경우)
      * 두 edge의 기울기가 같고, edge들의 직선의 방정식을 구했을 때 x값으로 0을 대입하여 얻은 값이 같으면 그 두 직선은 시작점이 같고 기울기도 같은 것으로 볼 수 있다. 이런 경우에는 교점이 무수히 많이 생기지만 우리의 알고리즘에서는 그 두 선분 중에서 짧은 선분의 가운데 있는 점을 교점으로 보아야 한다. 이유는 시작점이나 끝점중 하나는 항상 일치할 것이고, 우리가 고려할 사항은 그 edge위를 이동하는 경우에 중간점을 ‘거쳐서 가는가 거치지 않고 가는가’이기 때문이다.



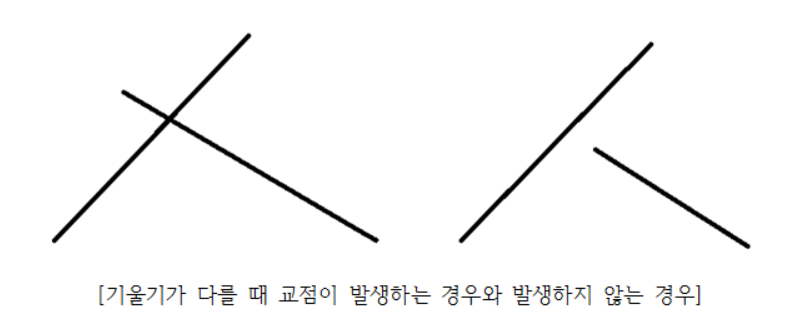
따라서 두 edge를 이루는 4개의 vertex중에서 가운데 오는 점이 어떤 vertex인지를 판별하여 그 vertex를 교점으로 보고, 그 교점을 지나는 두 edge를 저장한다.

* + - 일반적인 경우(두 edge의 기울기가 다르고 교점이 하나 발생하는 경우)
      * 교점의 x좌표와 y좌표를 각각 insx, insy라고 하자. 그리고 어느 한 edge에 대해서 그 edge를 포함하는 직선의 방정식을 y=(slope)x+displacement 라고 했을 때 교점의 좌표를 아래의 식에 의해서 구한다

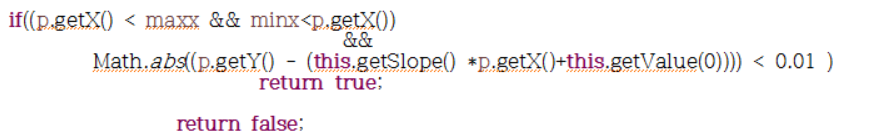
insx = -(displacement1 - displacement2)/(slope1 - slope2);

insy = (slope1\*displacement2 - displacement1\*slope2)/(slope1 – slope2);

그리고 edge1과 edge2를 이렇게 구한 교점을 vertex로 갖는 instance의 edgeset에 추가해 준다. 그런데 여기서 주의할 점은 기울기가 다르다고 해서 항상 교점이 생기는 게 아니라는 점이다. 기울기가 같을 때는, displacement만 비교해서 교점의 발생 여부를 알 수 있지만 기울기가 다른 경우에는 두 edge를 직선이라 생각하고 구한 교점이 실제로 그 선분들 위에 있는지 확인해야 한다.



이런 예외를 따로 처리하는 이유는 우리가 교점을 구하는 과정에서 실제로는 선분인 각 edge들을 직선의방정식으로 바꾸어서(즉, 직선으로 생각하여)교점을 구하기 때문이다. 그래서 위 그림에서 오른쪽 경우와 같이 실제로는 교점이 생기지 않지만 아래쪽에 있는 선분이 연장됐을 경우 생기는 교점을 일단 구하기 때문에, 그 교점이 실제로 각 선분위에 있는지를 아래의 식을 이용하여 판단한다.



여기서 값의 기준을 0.01로 잡은 것은 double의 계산에서 이 문제가 요구하는 소수점 아래의 수는 2자리이므로 그 자릿수에 영향을 주는 오차를 고려하기 위함이다.

* **Outline 출력 알고리즘**
* 본 코드에서 결과의 edge집합을 구하기 위해서 x,0위의 vertex중에서 x값이 가장 작은 vertex의 instance에서 시작하여 instance를 지나는 edge리스트들 중 기본적으로 기울기가 큰 순서대로 가지를 뻗어나가는 방식을 기반으로 한다. 여기에서 크게 두가지 고려사항이 있다. 첫 번째는 선택된 edge와 연장선은 동일하나 도메인이 다른 edge가 2개 이상 있을 때 어떤 점을 선택 할 것인지 이며, 두 번째는 기울기가 무한대인 직선이 해당 instance를 지날 때 무한대의 기울기를 가져서 다음 proper edge로서 우선순위는 가지지만 그 직선이 instance점 보다 y축으로 더 올라가지 않고 멈추는 경우 예를 들어instace (x,y)에서 다음 proper edge를 찾는데 instance를 지나는 edge리스트 중 edge { (x,0), (x,y) }가 있을 때 해당 edge는 가능한 리스트 중 가장 낮은 우선순위를 가져야 함에도 불구하고 무한대의 기울기로 가장 높은 우선순위를 가지게 되는 두 가지의 경우를 고려해야 했다.
* 사실 첫번째 경우는 알고리즘 설계 단계에서부터 예상했던 상황이었기 때문에 해당하는edge위의 instance들을instancelist클래스에 저장하고minDistance\_imp 메소드로 현재 접근중인 instance의 y좌표가 가장 높다면 그 다음으로 높은 instance를 proper edge로 삼고, 현재 인스턴스의 y 좌표가 가장 높지 않다면 후보들 중에서 그것보다 높되 y좌표가 가장 작은 인스턴스를 선택하는 방식으로 간단하게 해결하였다.
* 하지만 직각삼각형이 입력으로 들어오는 경우에 위 알고리즘을 적용하면 애초에 기울기를 구하는 부분에서부터 divide by 0에러를 만나게 되는 것으로 시작해서, 기울기 값을 사용해서 교점을 찾고, 또 traverse하는 과정에서 문제가 생기게 되었다. 따라서 edge클래스의 기울기를 저장할 때에 head와 tail의 x값이 같으면 기울기를 Double.MAXVALUE로 설정한 뒤 get\_properEdge\_Right메소드를 따로 만들어서 해당 instance를 지나는 기울기가 무한대인 선분에 대해 가장 끝나는 y값이 큰 edge를 반환하도록 하고, get\_properEdge는 예각의 edge중에서 기울기가 가장 큰 것을 반환하게 하도록 한 뒤, get\_properEdge\_imp에서 두 메소드의 리턴값을 받아서 둘중 하나라도 null이라면 비교할 필요 없이 null이 아닌 것을 그냥 리턴하고, 둘다 null이 아니라면 기울기가 무한대인 직선의 y값이 현제 instance의 y값보다 큰지 비교 한 후, 예각의 edge와 직각의 edge둘 중 하나를 선택한다. 코드는 아래와 같다.

**public** Edge get\_properEdge\_imp(Vertex v, Edge before){

Edge acute = **this**.get\_properEdge(v);

Edge right = **this**.get\_properEdge\_Right(v);

**if**(right == **null**) **return** acute;//예각만 있는 경우

**if**(acute == **null**) **return** right;//직각만 있는경우

**else**{//둘다 있는 경우, 직선의 y값이 더 위에 있으면 직선이 되고 직선이 점밑에 있으면 예각일떄를 고르는 것과 같다.

**if**(before.is\_right\_angle()) **return** acute;

**if**(right.bigYPos().getY()>v.getY()) **return** right;

**else** **return** acute;

}

}

* 위 과정을 반복하면 마지막으로 한가지만 더 고려하면 된다. 우리는 이 알고리즘이 매번 traverse상에 선택된 instance의 y좌표가 0일 경우에 (terminate)하도록 하였다. 인풋을 고려하였을 때 좌표평면 상에서 서로 연속되어 붙어있지 않은 도형이 나올 수도 있기 때문이다. 예를 들어 두 삼각형 {(0,0),(1,1),(1,0)}과 { (2,0), (3,1), (3,0) }과 같은 입력이 들어왔을 경우이다. 이 경우는 action 이라는 메소드를 만들어서 다루도록 하였다.

**public** Vertex action\_iter(Vertex iter)함수는 여태까지 설명한 outline출력 알고리즘 전체를 담고 있으며, 아래 코드의 메소드인 action이 action\_iter를 반복하면서 도형의 아웃라인을 덩어리단위로 그린다. 그러다가 마지막에 끝난 점이 변하지않고 terminate이면 루프를 끝내는 구조로 되어있다.

**public** **void** action(){

Vertex iter = **this**.start;

**while**(**true**){

Scanner s0 = **new** Scanner(System.***in***);s0.next();

iter = action\_iter(iter);

//한도형이 그려지고, 그것의 end point를 출력하는 식이 되야됨

Scanner s1 = **new** Scanner(System.***in***);s1.next();

//도형의 마지막 점이 output에 들어간다.

**if**(iter == terminate){

// 출력한 것이 terminate와 같다면 종료를 한다.

out.put(iter);

**break**;

}

**else**{//아니라면 새로운 점을 찾는다.

//바로 연결이 안된점이라면 넘어가기

**if**(!is.getInstance(iter).can\_go\_Right()){

out.put(iter);

iter = vs.get\_MinX\_0(iter.getX());

}

//연결이 되어있는 거면 계속 이어가되,두 번 들어가면 안되니 넣지는않아

}

}

1. **토의**

* **Big O-notation을 통한 분석**
* 우리가 사용한 알고리즘은 크게 두 부분으로 나눌 수 있다. 하나는 가능한 모든 교점을 구해 Instance를 형성하는 과정. 그리고 다른 하나는, 각 Instance들을 traverse하면서 Outline을 구해나가는 과정이다.
* **Instance 형성**
  + Instance란 기본적으로 입력 받은 모든 vertex와 그로 인해 생기는 edge들이 만나서 발생하는 교점들을 index로 갖는 객체이다. 각 instance에는 그 vertex를 지나는 edge들이 저장되어 있다. 이 Instance들을 구하기 위해 우리는 모든 edge에 각각에 대하여 다른 모든 edge들과 교점을 생성하는지 여부를 조사한다. 이 과정에서 edge를 시작점의 x좌표에 관하여 sort한 다음에 각 edge들을 그 종료 점의 x좌표보다 왼쪽에서 시작하는 edge들하고만 교점생성 여부를 조사하는 방법을 고려했으나(교점이 생기려면 적어도 현재 기준이 되는 edge의 끝점보다 왼쪽에서 시작해야 하므로), edgeset을 sort하는데 드는 비용을 고려하여 그냥 가능한 두 edge의 선택에 관하여 모두 조사하기로 하였다. 즉, 가능한 edge쌍의 개수는 n(n+1)/2 이고, 이 횟수만큼 교점발생 여부를 조사하게 된다.
* **Traverse**
  + 처음에는 모든 Instance들을 돌면서, 각 Instance에서 다른 모든 Instance들에 대해 edge로 연결되어 있는지 여부와, 연결되어 있다면 그들 중에서 가장 적절한(sky line을 이루는)edge는 어느 것인지 판단하여 그 vertex로 이동하는 알고리즘을 생각하였다. 그런데 이 알고리즘은 일단 Instance를 구하는데 n(n+1)/2의 연산 횟수가 들어간다. 그리고 각 Instance마다 다른 모든 Instance들에 대해서 조사를 해야 하므로, 대략 n(n+1)/2의 제곱만큼의 연산을 요구한다. 즉 O()이 되는 것이다. 그래서 자료를 좀 더 정리하여 각 Instance에 그 vertex를 지나는 edge정보를 추가하여, 그 Instance를 지나는 edge들에 대해서만 조사하는 알고리즘을 고안하였다.
* 우리가 사용한 알고리즘의 traverse과정을 다시 세분화하면 모든 Instance들을 iteration하는 과정과, 각 Instance에서 갈 수 있는 edge들 중에서 sky line을 만들기 위해 선택해야 하는 proper\_edge를 찾는 과정으로 나눌 수 있다. 위의 Instance형성 과정에서 봤듯이 총 Instance의 개수는 최대 n(n+1)/2개 일 것이다. 그리고 하나의 Instance가 생기기 위해서는 두 개의 edge가 만나야 한다. 만약에 한 Instance위에 여러 개의 edge가 동시에 지나가더라도 그건 그 vertex 한 점 위에 여러 개의 Instance가 겹쳐서 생성된 것이므로 그 각각의 Instance들은 2개의 edge를 자신을 지나는 선분으로 갖는다고 봐야 한다. 즉, 각 Instance가 proper\_edge를 찾을 때 고려해야 하는 edge의 수는 2로 볼 수 있고, 따라서 traverse과정에서 총 연산 횟수는 {n(n+1)/2} × 2 = n(n+1)이라고 볼 수 있다.
* **따라서 프로그램 전체의 Bin O notation은 O(n(n+1)/2 + n(n+1) = O(n^2) 이다.**

1. **구현과정에서 겪은 어려운 점**

* Vertex들을 구현하는데 있어서 같은 좌표 값을 가지지만 다른 객체로 되게 구현을 했었다. 이 과정에서, 원하는 구현결과가 나오지 않아 Set을 만들어 좌표 값에 대한 객체가 유일하게 하여 해결하였다.
* 직각 삼각형이 들어오는 것을 고려하지 않고 예각삼각형에 대해서 구현을 먼저 했었다. 후에 직각 삼각형도 생각해줘야 되는걸 알고 다시 구현하는데, 기존 코드에 직각인 경우에만 쓰는 method들을 합치면서 해결했다.
* 해당 Instance에 대해 적절한 Edge를 구하는 과정에서 여러 가지 경우의 수들이 발생했다. 직선이 들어올 때와 안 들어올 때에 대해 명확한 구분이 필요했기 때문이다. Test case들을 만들면서 주먹구구식으로 찾았는데, 이러한 과정을 통해 점차 Traverse 알고리즘을 개선해 나갔다.
* 적절한 Edge를 구한 뒤, 선택할 점을 찾는 것에도 애로사항이 있었다. 단순하게 가장 짧은 점을 구하는 식으로 생각을 했는데 test case를 돌려보며 안 되는 경우들이 발생했다. 그래서 기존 함수에 몇 가지 경우를 고려해 함수를 개선했다.