Notes on A6

Two parts:

learn(): Do a depth-first search of the whole graph, saving the state of each flyable tile. **Relatively easy**

Run(List<Long> idList): idList is a list of some of the flowers on the map. Some grew before learn() was called. Others grew after learn() was called; they are hard to find.

Purpose of run(). Have the Bfly fly around the map and collect the flowers who FlowerId is in idList.

Much harder, requiring more thought and design

Finish learn() VERY SOON. Make sure it is correct.

```
/** Node u is unvisited. Visit all nodes REACHABLE from u. */
public static void dfs(int u) {
    visited[u]= true;
    for each edge (u, v)
       if v is unvisited then dfs(v);
}
```

TRANSLATE THIS TO THE BFLY ENVIRONMENT

TileState[][] ts;   node u $\rightarrow$ ts[r][c]

0. The BFly does not necessarily start out on tile [0][0]!

1. A6 has wraparound!
E.g. East of tile ts[5][getMapWidth()-1] is tile ts[5][0].
Any index-expression must be calculated *mod* the width or height of the map.
Look in class Common for an existing mod function.

```
/** Node u is unvisited. Visit all nodes REACHABLE from u. */
public static void dfs(int u) {
    visited[u]= true;
    for each edge (u, v)
        if v is unvisited then dfs(v);
}
```

TRANSLATE THIS TO THE BFLY ENVIRONMENT

TileState[][] ts;   node u → ts[r][c]

2. General aim: Simplicity. As little case analysis as possible. Little duplication. Not too much loop nesting –use more methods

We urge you to write good, complete, precise method specs before writing the bodies. Reason: it allows YOU to write calls on methods without having to read method bodies. It makes programming some of the more complicated things easier.

```
/** Node u is unvisited. Visit all nodes REACHABLE from u. */
public static void dfs(int u) {
    visited[u]= true;
    for each edge (u, v)
        if v is unvisited then dfs(v);
}
```

TRANSLATE THIS TO THE BFLY ENVIRONMENT

TileState[][] ts; node u → ts[r][c]

3. Urge you to make: visited[u] = ts[r][c] != null

To make run() easier later on, store an object in a cliff or water tile when encountered. See static variables in class TileState

4. Does the BFly DFS procedure need parameter u, or can it be given by the Bfly's current tile? Think about this, remove parameter if unnecessary.

```
/** Node u is unvisited. Visit all nodes REACHABLE from u. */
public static void dfs(int u) {
      visited[u]= true;
      for each edge (u, v)
         if v is unvisited then dfs(v);
}
```

TRANSLATE THIS TO THE BFLY ENVIRONMENT

TileState[][] ts;   node u $\rightarrow$ ts[r][c]

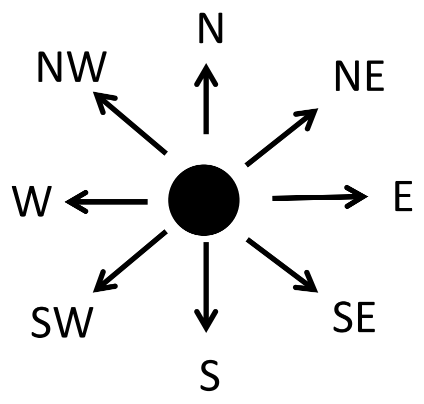
5. Public static void dfs() {…}

Where should the Bfly end up after completing the dfs? Think carefully about that and put it into the specification.

```
/** Node u is unvisited. Visit all nodes REACHABLE from u. */
public static void dfs(int u) {
    visited[u]= true;
    for each edge (u, v)
       if v is unvisited then dfs(v);
}
```

TRANSLATE THIS TO THE BFLY ENVIRONMENT

6. The Bfly can fly in 8 possible directions.
Study class Directions!



7. How to translate **for each edge** (u, v) {…}
Into a simple statement

for (*type variable* : *expression*) {…}

that sequences through the 8 Directions?

```
/** Collect the flowers whose FlowerIds are in list fList.
    Don't collect any other flowers. Bfly must visit fewer nodes
    than it did during learn() */
public void run(List<Long> fList)
```

fList: (3, 6, 10, 11, 12, 30, 40, 41)

Some flowers were there during learn. You can find which tile they are on simply by searching through ts[][] (but to collect a flower, you have to fly to the tile it is on)

Some flowers were planted after learn. You can't find one in the states of ts[][] until you fly to the tile it is on and refresh

Use flower aromas to find direction to fly to for a flower.

```
/** Collect the flowers whose FlowerIds are in list fList.
    Don't collect any other flowers. Bfly must visit fewer nodes
    than it did during learn() */
public void run(List<Long> fList)
```

fList: (3, 6, 10, 11, 12, 30, 40, 41)

flower aroma on a tile spreads –further away, less intense.

You have to use the aroma to find the tile with the flower: to fly one step closer to the flower, fly to a tile where its aroma is higher.

To find a tile's intensity, the Bfly must be on the tile.

25	25	25	12
25	100	25	12
25	25	25	12
12	12	12	12

Bfly's state contains:
List<aroma>
List<Flower>

An aroma contains:
Field intensity
Function getFlower()

```java
/** Collect the flowers whose FlowerIds are in list fList.
    Don’t collect any other flowers. Bfly must visit fewer nodes
    than it did during learn() */
public void run(List<Long> fList)
```

25	25	25
25	100	25
25	25	25

fList: (3, 6, 10, 11, 12, 30, 40, 41)

Different strategies. Here are two examples, both using shortest path and a method M that flies to a tile using a flower aroma.

1. Do shortest path algorithm. Then find all old flowers using its shortest path and new flowers using M. To use shortest path, requires always returning to starting point.

2. Do shortest path algorithm. Then find all old flowers using its shortest path and new flowers using M. To use shortest path, requires always returning to starting point.

Can you find strategies that don’t always require going back to initial node —perhaps using shortest path algorithm more than once? But change it so it doesn’t compute everything