Problem 1: Publishing Stories

Social networking sites like Facebook, LinkedIn, and Google+ typically record and publish stories about actions taken by you and your friends. Stories like:

Jessie Duan accepted your friend request.
Matt Anderson is listening to Green Day on Spotify.
Patrick Costello wrote a note called "Because Faiz told me to".
David Wang commented on Jeffrey Spehar’s status.
Mike Vernal gave The French Laundry a 5-star review.

are created from story templates like

{name} accepted your friend request.
{name} is listening to {band} on {application}.
{name} wrote a note called "{title}".
{name} commented on {target}'s status.
{actor} gave {restaurant} a {rating}-star review.

The specific story is generated from the skeletal one by replacing the tokens—substrings like "{name}", "{title}", and "{rating}"—with event-specific values, like "Jessie Duan", "Because Faiz told me to", and "5". The token-value pairs can be packaged in a Map<string, string>, and given a story template and a data map, it's possible to generate an actual story.

Write the generateStory function, which accepts a story template (like "{actor} gave {restaurant} a {rating}-star review." ) and a Map<string, string> (which might map "actor" to "Mike Vernal", "restaurant" to "The French Laundry", and "rating" to "5"), and builds a string just like the story template, except the tokens have been replaced by the text they map to.

Assume the following is true:

- '{' and '}' exist to delimit token names, but won’t appear anywhere else. In other words, if you encounter the '{' character, you can assume it marks the beginning of a token that ends with a '}'.
- We guarantee that all tokens are in the Map<string, string>. You don’t need to do any error checking.
The prototype is:

```cpp
static string generateStory(const string& storyTemplate, const Map<string, string>& data);
```

**Problem 2: Topswopping and Topswop Numbers**

When a `Stack<int>` of depth `n` contains the numbers 1 through `n` in some order, its **Topswop number** is defined to be the number of times the top of the stack must be **swopped** before the top element becomes a 1. Each swop amounts to an examination of the topmost element of the stack—let’s call it `k`—and then the inversion of the top `k` elements. (A stack whose top element is already 1 has a Topswop number of 0.)

```
2
5
1
3
4
```

Implement the `getTopswopNumber` function, which accepts a copy of a `Stack<int>` and returns the number of times the `Stack<int>` must be swopped before the top element becomes a 1. You may assume the `Stack<int>` contains the numbers 1 through `n` in some order, and you can trust the fact that the process always terminates.

```cpp
static int getTopswopNumber(Stack<int> s);
```
Problem 3: Chain Reactions

Chain Reaction is a one-player game that made its way through the Internet and various mobile platforms some eight or nine years ago. In our version, we’ve given a collection of locations—GPoints in the plane—of land mines. The detonation of any single land mine prompts all land mines within a certain distance to simultaneously detonate a second later, which themselves set off more land mines another second later, and so forth. The chain reaction continues until there are no active land mines, or until none of the remaining land mines fall within the threshold distance of those that’ve already exploded.

- You get 0 points for the land mine you initially [and manually] detonate.
- You get 100 points for each land mine that detonates at the one-second mark.
- You get 400 points for each land mine that detonates at the two-second mark.
- You get 900 points for each land mine that detonates at the three-second mark.
- In general, you get $100n^2$ points for each land mine that detonates at the $n$-second mark.

Implement the `computeAllScores` function, which accepts a reference to a `Set<GPoint>` of all the land mine locations, and populates the referenced `Map<GPoint, int>`—assumed to be empty when `computeAllScores` is called—with the score attained by manually detonating the land mine at each of the locations.

Specifically, each `GPoint` in the `Map<GPoint, int>` should map to the score one would get by detonating it before all others. For simplicity, assume that `operator<` has already been defined so that `GPoints` can be stored as keys in `Maps` and as entries in `Sets`.

```cpp
static void computeAllScores(const Set<GPoint>& landMines,
                              Map<GPoint, int>& scores);
```

Further assume that the following convenience function—the predicate function that determines if one mine is close enough to a second to detonate it one second later—has been provided as well.

```cpp
const static double kThresholdDistance = 50; // in pixels
static bool isInRange(const GPoint& base, const GPoint& target) {
    double dx = base.getX() - target.getX();
    double dy = base.getY() - target.getY();
    return dx * dx + dy * dy <= kThresholdDistance * kThresholdDistance;
}
```

Problem 4: Stepping Stones

Stepping stone puzzles are grids of colored circles where the goal is to travel from the `start` stone to the `finish` stone by stepping up, down, left, and right. As you travel, you must visit three stones of the same color, switch color, visit three stones of another single color, switch color, and so on. You may not make any U-turns (that is, you’re not allowed to back up onto a stone that you most recently came from), but you’re otherwise allowed to visit the same stone several times. The initial step off of `start` must change color, and
your final step onto **finish** must change color as well (and you can step on the **start** and **finish** stones along the way if it helps).

If, for instance, you are presented with the stepping stones below (where different fill patterns represent different colors), you can navigate from the upper left corner—coordinate (0, 0)—to the lower right corner—coordinate (4, 4)—in a breezy 34 steps.

![Stepping Stones Diagram](image)

Travel starts out like this:

![Travel Starts](image)

and ends like this:

![Travel Ends](image)

Implement `generatePath`, which uses breadth-first-search to find the **shortest sequence of stones** one must step on to get from **start** to **finish**. The puzzle is modeled as a `Grid<string>`, where the strings are the colors, spelled out like "**Yellow**" and "**Green**". You can assume that a solution is known to exist, and you needn’t avoid cycles while doing the search (knowing they won’t be present in the shortest path solution.
anyway). You may assume you’ve access to the following type definition and helper function:

```cpp
struct stone {
  int row;
  int col;
};

static stone getNeighboringStone(const stone& location, Direction dir) {
  stone neighbor = location;
  switch (dir) {
    case NORTH: neighbor.row--; break;
    case EAST: neighbor.col++; break;
    case SOUTH: neighbor.row++; break;
    case WEST: neighbor.col--; break;
  }
  return neighbor;
}
```

Assume that <, ==, and != have been overloaded so you can compare stones. The shortest path—expressed as a `Vector<stone>` should include `start` at the front and `finish` at the back, and should be written in the space referenced by `shortest`.

```cpp
static void generatePath(const Grid<string>& stones, const stone& start, const stone& finish, Vector<stone>& shortest);
```