CS106B Handout 15
Autumn 2012 October 8th, 2012

Section Handout

Discussion Problem 1: Chain Reactions

Chain Reaction is a popular one-player game that recently made its way through the Internet and various mobile platforms. In our version, we've given a collection of locations—**`GPoint`**s 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 as **`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 **`GPoint`**s can be stored as keys in **`Map`**s and as entries in **`Set`**s.

```
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.

```
const static double kThresholdDistance = 50;
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;
}
```

Discussion Problem 2: GCD [CS106B Reader, Chapter 7, Problem 4]

The **greatest common divisor** (often abbreviated to **gcd**) of two nonnegative integers is the largest integer that divides evenly into both. In the third century BCE, the Greek mathematician Euclid discovered that the greatest common divisor of **x** and **y** can always be computed as follows:

- If **x** is evenly divisible by **y**, then **y** is the greatest common divisor.
- Otherwise, the greatest common divisor of **x** and **y** is always equal to the greatest common divisor of **y** and the remainder of **x** divided by **y**.

Use Euclid's insight to write a recursive function **gcd** that computes the greatest common divisor of **x** and **y**.

Discussion Problem 3: Twiddles

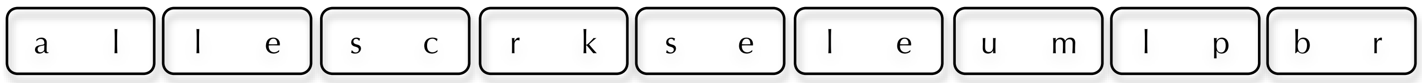
Two English words are considered *twiddles* if the letters at each position are either the same, neighboring letters, or next-to-neighboring letters. For instance, **sparks** and **snarls** are twiddles. Their second and second-to-last characters are different, but **p** is just two past **n** in the alphabet, and **k** comes just before **l**. A more dramatic example: **craggy** and **eschew**. They have no letters in common, but **craggy**'s **c**, **r**, **a**, **g**, **g**, and **y** are **-2**, **-1**, **-2**, **-1**, **2**, and **2** away from the **e**, **s**, **c**, **h**, **e**, and **w** in **eschew**. And just to be clear, **a** and **z** are **not** next to each other in the alphabet—there's no wrapping around at all.

Implement a recursive procedure called **listTwiddles**, which accepts a reference to a string **str** and a reference to an English language **Lexicon**, and prints out all those English words that just happen to be **str**'s twiddles. You'll probably want to write a wrapper function. (Note: any word is considered to be a twiddle of itself, so it's okay to print it.)

```
static void listTwiddles(const string& str, const Lexicon& lex);
```

Lab Problem 1: Letter Rectangles and Words

You are given a large collection of short, fat rectangles, where each half of each rectangle contains a single letter, as with:



Given the option to rearrange, ignore, and rotate pieces, you're charged with the task of identifying all of the even-length English words that can be formed by chaining together some subset of the pieces (where some may have been rotated). For the above set of pieces, the list of printed words should surely include **"plum"**, since the third-to-last rectangle can be placed after the second-to-last rectangle (rotated so that the **'p'** precede the **'l'**) to form **"plum"**. Given the above set of rectangles, you should also identify fun words like **"allele"**, **"lark"**, **"muscle"**, **"scales"**, and **"umbrella"**, in addition to quite a few others. Note that each rectangle can be used at most one time per word, so that words like **"sees"** and **"museum"** can't be formed.

Using the lab starter files up on the course web site, work in pairs (and get help as needed) to implement the recursive function **gatherWords**, which accepts references to a **Vector<string>** called **rects** (where each **string** is two characters), a **Lexicon** constant called **english**, and an initially empty **Lexicon** called **words**, and populates **words** with the collection of those words, and only those words, that can be formed using the rectangles in **rects**. You should implement this using a wrapper function.

```
   static void gatherWords(const Vector<string>& rects,
                           const Lexicon& english, Lexicon& words);
```