CS106X
Autumn 2012

Handout 25
October 23rd, 2012

CS106X Midterm Examination

This is closed book, closed notes, closed reader, closed everything exam. If you're taking the exam remotely, you can telephone Jerry at 415-205-2242 while taking the exam to ask questions.

Good luck!

Section Leader: ____________________

Last Name: ____________________

First Name: ____________________

I accept the letter and spirit of the honor code.

(signed) ____

		Score	Grader
1. Stepping Stones	[15]	______	______
2. Multitions	[15]	______	______
3. Valency	[15]	______	______
Total	**[45]**	______	______

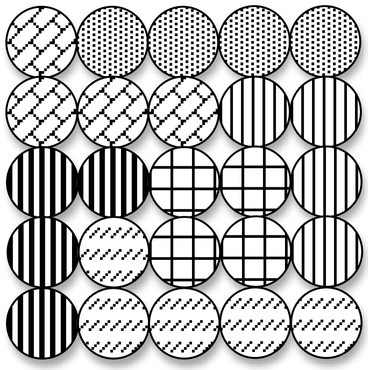
Summary of Relevant Data Types

```
class string {
   bool empty() const;
   int size() const;
   int find(char ch) const;
   int find(char ch, int start) const;
   string substr(int start) const;
   string substr(int start, int length) const;
   char& operator[](int index);
   const char& operator[](int index) const;
};

enum Direction { NORTH, EAST, SOUTH, WEST };

class Vector {
   bool isEmpty() const;
   int size() const;
   void add(const Type& elem); // operator+= used similarly
   void insert(int pos, const Type& elem);
   void remove(int pos);
   Type& operator[](int pos);
};

class Grid {
   int numRows() const;
   int numCols() const;
   bool inBounds(int row, int col) const;
   Type get(int row, int col) const; // cascade of operator[] also works
   void set(int row, int col, const Type& elem);
};

class Stack {
   bool isEmpty() const;
   void push(const Type& elem);
   Type pop();
};

class Queue {
   bool isEmpty() const;
   void enqueue(const Type& elem);
   Type dequeue();
};

class Map {
   bool isEmpty() const;
   int size() const;
   void put(const Key& key, const Value& value);
   bool containsKey(const Key& key) const;
   Value get(const Key& key) const;
   Value& operator[](const Key& key);
};

class Set {
   bool isEmpty() const;
   int size() const;
   void add(const Type& elem); // operator+= also adds elements
   bool contains(const Type& elem) const;
};
```

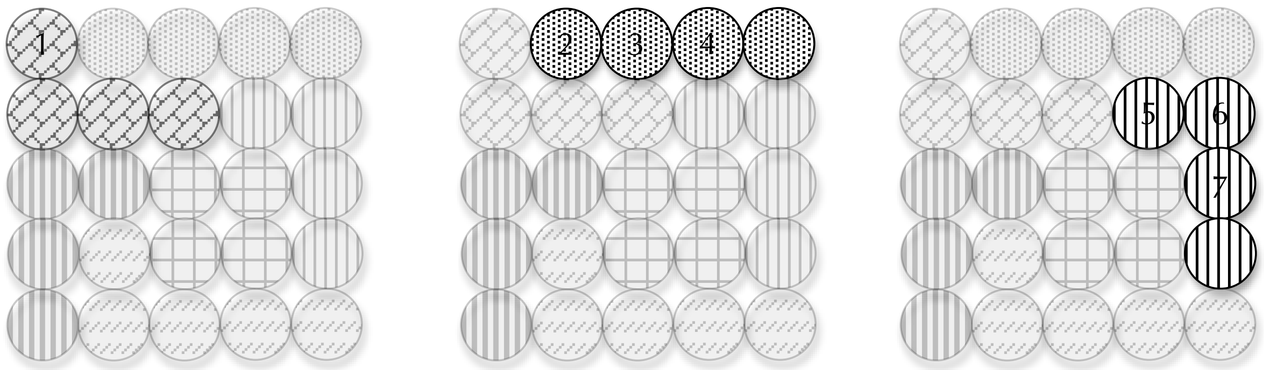
Problem 1: Stepping Stones [15 points]

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 (although you can step on the **`start`** and **`finish`** stones along the way if it helps).

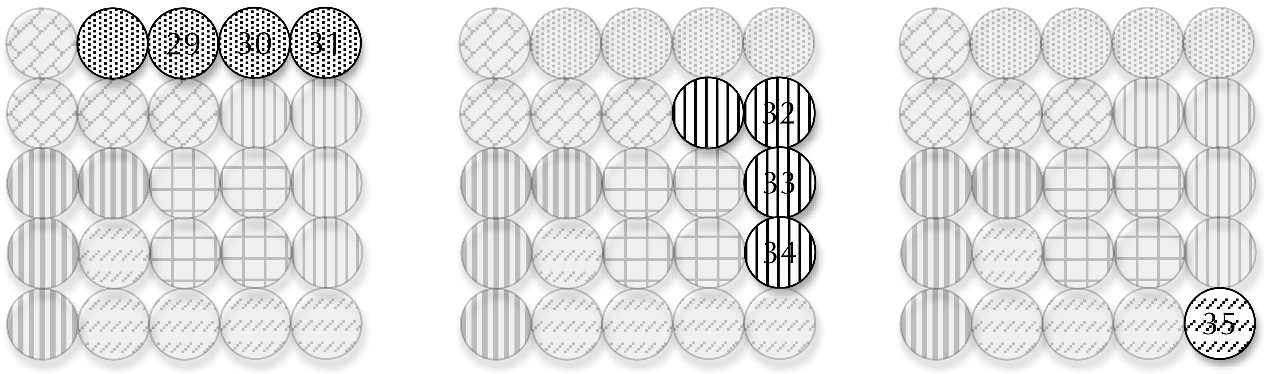
If, for instance, you are presented with the stepping stones below (where different fill patterns represent different colors), you can navigate from the upper left corner—coordinate (0, 0)—to the lower right corner—coordinate (4, 4)—in a breezy 34 steps.



Travel starts out like this:



and ends like this:



Complete the implementation of a function called **`generatePath`**, which uses breadth-first-search (as your **`word-ladder`** solution did) 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). You may further assume you've access to the following type definition and helper function:

```
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 **`stone`**s. 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`**. Use the next page to present your implementation. (Note that you **should not use recursion** for this problem. You must use a genuine breadth-first search approach and generate all paths of length 1, then all paths of length 2, and so on, until you generate a path that incidentally leads to the target stone with the right modulo-3 properties).

Problem 1: Stepping Stones [continued]

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

Problem 2: Multitions [15 points]

A multition of order **n** is the insertion of **n** multiplication signs in between arbitrary digits of a number so the result is a valid arithmetic expression. Here are just some of the order-2 multitions of the number 234567898765432:

```
2345 * 678987 * 65432
2 * 345678987654 * 32
23 * 4567898765 * 432
23456 * 789 * 8765432
```

Each of these expressions, when evaluated, yields different results:

```
2345 * 678987 * 65432  =  104182434465480
2 * 345678987654 * 32  =  22123455209856
23 * 4567898765 * 432  =  45386642129040
23456 * 789 * 8765432  =  162219956690688
```

Present your implementation of **`optimalMultition`**, which **recursively generates** all order-**`n`** multitions of the specified **`number`**, returns the largest product of all multition products, and updates **`factors`** to contain the multitioned factors contributing to the maximum product. **`optimalMultition(23436234, 3, factors)`**, for instance, would return 2464588 and populate the **`Stack`** referenced by factors with the numbers 23 (at the top of the stack), 43, 623, and 4 (at the bottom of the stack). If there are no order-**`n`** multitions of the provided number, your function should return 0 and leave the stack empty. You may also assume that the **`int`** type can store arbitrarily large integer values.

Present your implementation on the next page (and feel free to tear this page out so you can easily refer to it). In addition to the standard string methods presented on page 2, you will benefit from using the following functions from **`strlib.h`**:

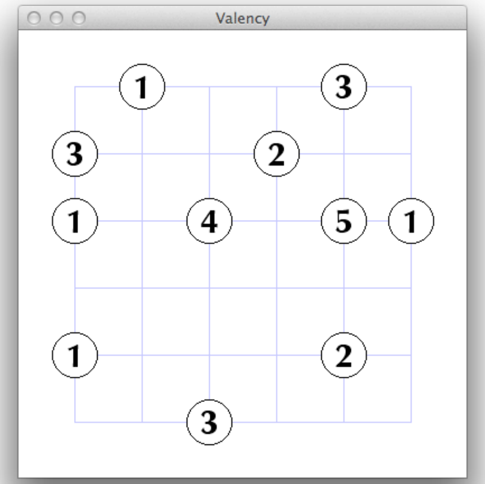
```
string integerToString(int n);
int stringToInteger(const string& str);
```

Problem 2: Multitions [continued]

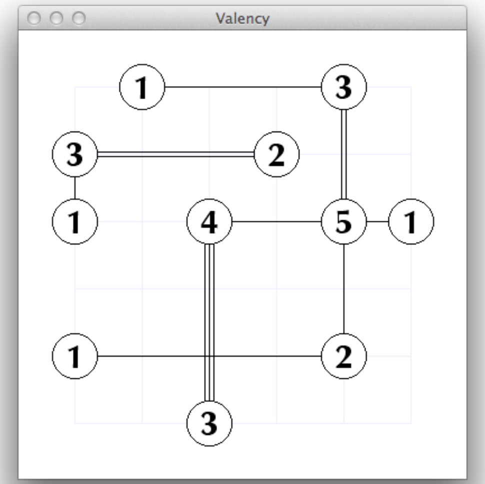
```
static int optimalMultition(int number, int count, Stack<int>& factors) {
```

Problem 3: Valency [15 points]

Valency is a puzzle one solves by repeatedly connecting pairs of circles. Any two circles can be vertically or horizontally linked one or more times, provided there are no other circles in between them. Each circle has an associated **valency** specifying the exact number of connections one must draw between it and other circles. The goal of the puzzle is to connect circles to one another so that all valency constraints are satisfied.



One such puzzle is presented on the left, and one of its solutions is presented on the right. This particularly solution makes use of 1 triple, 2 double, and 6 single connections—a total of 13 in all—to satisfy a combined valency constraint of 26.



The puzzle can be modeled as a **`Grid<int>`**, where a zero reflects the absence of a circle, and a positive value reflects the presence of one.

To help simplify the problem, you should rely on the services of a few data types and helper functions. In particular, you should assume that the following two data types have been defined for you and that operators like **<** have been overloaded so that each may be stored as entries in **`Set`**s and as keys in **`Map`**s.

```
struct coord {            struct connection {
   int row;                  coord first;
   int col;                  coord second;
};                        };
```

You can also assume the following two functions have already been implemented for you:

```
static int computeValencySum(Grid<int>& valencies);
static Set<coord> getCandidates(const coord& location, Grid<int>& valencies);
```

`computeValencySum` returns the sum of all of the supplied **`Grid`**'s entries, and **`getCandidates`** returns the **`Set<coord>`** of all other circles with **nonzero** valency that **`location`** could potentially be connected with given the state of the supplied **`Grid`**.

Implement a recursive backtracking **`solve`** routine that returns **`true`** if and only if the referenced Valency puzzle—encoded as a **`Grid<int>`** by the name of **`valencies`**—can be solved. When **`true`** is returned, the referenced **`connections`** should contain all of the connections (mapped to their multiplicity) that solve the puzzle. For the puzzle presented above, **`solve`** should return **`true`** and update the referenced **`Map`** with 9 **`connection`**s as keys. 6 of the 9 **`connection`**s should map to 1 (because 6 of the nine pairings are singly connected), 2 of the 9 should map to 2 (because two pairs are doubly connected), and the 9th **`connection`** should map to a 3. (Be sure to remove any temporarily inserted **`connection`**s that ultimately map to 0.) If **`false`** is returned, then the state of the referenced **`Map`** can be ignored. Use the rest of this page and the next to present your answer.

```
   static bool solve(Grid<int>& valencies, Map<connection, int>& connections) {
```

Problem 3: Valency [continued]