This is an open-book, open-note, closed-electronic-device exam. You have 80 minutes to complete the exam. If you’re taking the exam remotely, you can telephone Jerry at 415-205-2242 to ask questions.

Good luck!

Section Leader: _______________________

Last Name: _______________________

First Name: _______________________

SUNet ID: ________________________@stanford.edu

I accept the letter and spirit of the honor code.

(signed) __________________________________________________________

<table>
<thead>
<tr>
<th>Question</th>
<th>Maximum Score</th>
<th>Score</th>
<th>Grader</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Total</strong></td>
<td><strong>[40]</strong></td>
<td>______</td>
<td>______</td>
</tr>
</tbody>
</table>
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;
};

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);
    const Type& operator[](int pos) const;
};

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

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

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;
};

class Lexicon {
    bool contains(const string& str);
    bool containsPrefix(const string& str);
    void add(const string& word);
};
Problem 1: Word Ladders, Take II [15 points]

For Assignment 2, you implemented a breadth-first search algorithm that generates the shortest word ladder between two words. The pseudo-code presented in the assignment handout was this:

```cpp
create initial ladder (just start word) and enqueue it
while queue is not empty
    dequeue first ladder from queue (this is shortest partial ladder)
    if top word of this ladder is the destination word
        return completed ladder
    else for each word in lexicon that differs by one char from top word
        and has not already been used in some other ladder
        create copy of partial ladder
        extend this ladder by pushing new word on top
        enqueue this ladder at end of queue
```

An implementation coded to specification never uses a previously used word to extend a partial ladder. Stated differently, each word—whether or not it ultimately contributes to the word ladder of interest—has a unique predecessor.

[5 points] Imagine that you have access to a `Map<string, string>` called `predecessors`. Each key maps to the word preceding it in all partials ever generated during a search. Given references to the `start` word, the `finish` word, and this `predecessors` map, it’s possible to reconstruct and return the word ladder connecting `start` to `finish`. Implement the `reconstruct` function, which does exactly that. Use the rest of this page to present your implementation.

```cpp
static Vector<string> reconstruct(const string& start, const string& finish, const Map<string, string>& predecessors) {
```

Your Assignment 2 implementation made use of a `Queue<Vector<string>>` to maintain a first-in-first-out list of all the partials ever generated during a search. It’s possible to reduce the memory footprint of the breadth-first search by relying on a `Queue<string>` (where each `string` is the last word of a partial word ladder), provided you maintain a `predecessors` map along the way as well. Restated, it isn’t necessary to (and for this problem you shouldn’t) maintain a queue of partial word ladders, since all partial word ladders are implied by their last word and the information in a `predecessors` map.

[10 points] Using the `reconstruct` function from the previous page, implement the `findShortestWordLadder` function, which accepts references to `start` and `finish` (you can assume they’re each `strings` of the same length), and returns the shortest word ladder between them (or the empty `Vector<string>` if there isn’t one). You should rely on the following function (you may assume it has already been implemented for you):

```cpp
static Vector<string> generateAllNeighbors(const string& word, const Lexicon& english);
```

which returns all English words that differ from the provided one by exactly one letter.

Use this page and the next to present your `findShortestWordLadder` implementation.

```cpp
static Vector<string> generateShortestWordLadder(const string& start, const string& finish, const Lexicon& english) {
```
Problem 1: Word Ladders, Take II [continued]
Problem 2: Matryoshkas [15 points]

Matryoshkas are sets of traditional Russian wooden dolls of decreasing size placed one inside the other. A matryoshka doll can be opened to reveal a smaller figure of the same sort inside, which has, in turn, another figure inside, and so on.

The Matryoshka Museum in Moscow recently exhibited a collection of similarly designed matryoshka sets, differing only in the number of nested dolls in each. Unfortunately, some overly zealous children separated these sets, placing all the individual dolls in a row. There are \( n \) dolls in the row, each with an integer size. You need to reassemble the matryoshka sets, knowing neither the number of sets nor the number of dolls in each set. You know only that every complete set consists of dolls with consecutive sizes from 1 to some number \( m \), which may vary between the different sets.

When reassembling the sets, you must follow these rules:

- You can put a doll or a nested group of dolls only inside a larger doll.
- You can combine two groups of dolls only if they are adjacent in the row.
- Once a doll becomes a member of a group, it cannot be transferred to another group or permanently separated from the group. It can be temporarily separated only when combining two groups.

Obviously, you want to reassemble the matryoshkas as quickly as possible. The only time-consuming part of reassembly is opening a doll, so you want to minimize how often you do this. For example, the minimum number of openings (and subsequent closings) when combining a set \{1, 2, 6\} with the group \{4\} is two, since you have to open the dolls with sizes 6 and 4. When combining a set \{1, 2, 5\} with the group \{3, 4\}, you need to perform three openings, since you need to open 3, which means you need to open 4, which means to need to open 5.

Each matryoshka set is modeled as a `Set<int>`, and the initial row of dolls can be modeled as a `Vector<Set<int>>`, where every `Set<int>` in the initial `Vector` is a singleton representing a single matryoshka doll. Your job here is to write a recursive function called `minimumOpensNeeded` that computes the minimum number of times all dolls needs to be opened in order for the full set of matryoshkas to be fully reassembled.
While implementing the recursive function, you may assume the existence of these helper functions:

```cpp
static bool allSetsAreComplete(const Vector<Set<int>>& sets);
static int mergeCost(const Set<int>& one, const Set<int>& two);
```

The first function returns `true` if and only every set in the vector represents a complete matryoshka set, and the second returns the minimum number of opens needed to merge the two sets (or `INT_MAX` if the two sets can’t be merged).

Use the rest of the page and the next one to implement `minimumOpensNeeded`. You may assume that each of the sets in the initial vector of sets is a singleton. If the original vector of `Set<int>`s can’t be merged into one or more complete sets, then it should return `INT_MAX`. Do not try to incorporate any memoization.

Sample calls:

```cpp
minimumOpensNeeded({{1},{2},{3},{2},{4},{1},{3}}) returns 7.
minimumOpensNeeded({{1},{2},{1},{2},{4},{3},{3}}) returns INT_MAX.
minimumOpensNeeded({{1},{1},{1},{1},{1},{1},{1}}) returns 0.
minimumOpensNeeded({{1},{2},{1},{2},{1},{2},{1}}) returns 4.
```

```cpp
static int minimumOpensNeeded(Vector<Set<int>>& sets) {
```
Problem 2: Matryoshkas [continued]
Problem 3: Short Answer Questions [10 points]

Unless otherwise noted, your answers to the following questions should be 75 words or fewer. **Responses longer than the permitted length will receive 0 points.** You needn’t write in complete sentences provided it’s clear what you’re saying. Full credit will only be given to the best of responses. Just because everything you write is true doesn’t mean you get all the points.

a) [2 points] When passing a large data structure, we sometimes pass a copy, and other times we pass it by `const` reference. Briefly describe why a helper function might prefer a copy (e.g. `foo(Map<string, string> m)`) instead of a `const` reference (e.g. `foo(const Map<string, string>& m)`, even though it’s more expensive.

b) [2 points] Any algorithm implemented using a **Queue** could be rewritten using a **Vector** instead. List two distinct advantages of the **Queue** over the **Vector** if an algorithm can be implemented using either one.
c) [2 points] Recall that your Boggle assignment relied on the **Lexicon** class, and in particular, relied on its `contains` and `containsPrefix` method. How would your implementation of Boggle have been impacted had the **Lexicon** not included `containsPrefix`?

d) [2 points] The last problem from Assignment 4 required you use recursion to compute the minimum number of votes one could receive and still win the Electoral College. A working solution needed to employ memoization in order to run quickly.

How does the fact that so many states have the same number of electoral votes (more than half of the 50 states have between 3 and 9 electoral votes, inclusive) impact the size of the memoization cache?
e) [2 points] One of the sample calls I provided for Problem 2 was

\[
\text{minimumOpensNeeded}([[1],[2],[1],[2],[4],[3],[3]]) \text{ returns INT_MAX.}
\]

A particularly robust implementation would have validated the provided input before committing to any recursion and much more quickly determined that the two matryoshka sets couldn’t be fully restored.

Briefly describe how you could quickly identify whether or not the supplied input is valid. Your description should be able to identify any invalid input, and not just the one in the above sample call.