Hemanth, Annie, Travis, and Jack all graded your exams over the weekend, and they’re now available online (we scanned all of them for a distributed grading party) at www.gradescope.com under your SUNet ID (e.g. poohbear@stanford.edu).

The midterm was challenging, and I managed to require nontrivial use of virtually all of the container classes between the two coding problems. Midterm scores ranged from 6 all the way up to a 36, the median was a 22, and the standard deviation was 8.2. The median was lower than it’s been in past quarters, but I credit that to a slightly different exam format (only two coding questions instead of three, and short answer questions for the first time ever), the demanding recursion problem, and the fact that the midterm was 80 minutes long instead of two hours. So that grades are consistent with what they’ve been in prior quarters, the median grade will be curved up to an 80%, the highest grade will be curved up to a 100%, and everything else will be scaled up proportionally in that familiar $y = mx + b$ manner (36 ⇒ 100%, 29 ⇒ 90%, 22 ⇒ 80%, 15 ⇒ 70%, 8 ⇒ 60%, and so forth).

If you have questions about how your midterm was graded, or you feel some of your work was overlooked, you’re more than welcome to request a regrade. All regrade requests must go through Jerry, however, and you must come see me in person during office hours or during some time scheduled outside of my normal hours. Please don’t be shy about asking for a regrade request, since the exam is too large a part of your final grade, and I want everyone moving forward trusting his or her exam was graded properly.

<table>
<thead>
<tr>
<th>Score</th>
<th>Grader</th>
</tr>
</thead>
<tbody>
<tr>
<td>1. Word Ladders, Take II</td>
<td>[15]</td>
</tr>
<tr>
<td>3. Short Answer Questions</td>
<td>[10]</td>
</tr>
<tr>
<td>Total</td>
<td>[40]</td>
</tr>
</tbody>
</table>
Solution 1: Word Ladders, Take II [15 points, Median Grade: 10]

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:

```plaintext
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.

```c++
static Vector<string> reconstruct(const string& start, const string& finish, const Map<string, string>& predecessors) {
    Stack<string> inverted; // could have been a Vector as well
    string rung = finish;
    while (true) {
        inverted.push(rung);
        if (rung == start) break;
        rung = predecessors[rung];
    }
    Vector<string> ladder;
    while (!inverted.isEmpty()) {
        ladder += inverted.pop();
    }
    return ladder;
}
```

**Problem 1 reconstruct Criteria: 5 points**

- Correctly reaches every single intermittent word in the ladder via the `Map`: 2 points
- Correctly includes both endpoints of the ladder: 1 point
- Correctly lays down all of the rungs of the word ladder in the correct order: 2 points
  (be sensitive to the possibility the student may have searched from finish to start, so that predecessors are effectively successors)
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) {
    Map<string, string> predecessors;
    Queue<string> queue;
    queue.enqueue(start);
    predecessors[start] = "";
    while (!queue.isEmpty()) {
        string endpoint = queue.dequeue();
        if (endpoint == finish) return reconstruct(start, finish, predecessors);
        Vector<string> neighbors = generateAllNeighbors(endpoint, english);
        for (const string& neighbor: neighbors) {
            if (!predecessors.containsKey(neighbor)) {
                predecessors[neighbor] = endpoint;
                queue.enqueue(neighbor);
            }
        }
    }
    return Vector<string>();
}
```
Problem 1 `generateShortestWordLadder` Criteria: 10 points

- Properly declare the predecessors map, the queue of partials, and populate the queue of partials with the `start` (or `finish`) word: 1 point
- Takes whatever measures necessary to ensure that the initial word doesn’t get recorded as a predecessor (or is at least recognized as a special case somewhere): 1 point
- Properly loops until queue is drained: 1 point
- Returns the empty vector if the queue is ever drained and no word ladder was ever found: 1 point
- Dequeues the leading representative of the best candidate partial: 1 point
- Returns the reconstructed word ladder if that leading representative if the `finish` (or `start`) word: 1 point
- Generates and correctly iterates over all neighbors via `generateAllNeighbors`: 1 point
- Excludes all those that have been seen before by checking to see if a predecessor has been recorded: 1 point
- Updates the `predecessors` map accordingly: 1 point
- Appends the new neighbor to the end of the queue: 1 point
**Solution 2: Matryoshkas [15 points, Median Grade: 6]**

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) {
    if (allSetsAreComplete(sets))
        return 0;

    int cost = INT_MAX;
    for (int i = 0; i < sets.size() - 1; i++) {
        int init = mergeCost(sets[i], sets[i + 1]);
        if (init != INT_MAX) {
            Set<int> second = sets[i + 1];
            sets[i] += sets[i + 1];
            sets.remove(i + 1);
            int rest = minimumOpensNeeded(sets);
            if (rest != INT_MAX && init + rest < cost)
                cost = init + rest;
            sets[i] -= second;
            sets.insert(i + 1, second);
        }
    }

    return cost;
}
```
Problem 2 Matryoshkas Criteria: 15 points

- Identifies the base case scenario: 2 points
  - Properly identifies the base case scenario by calling `allSetsAreComplete`: 1 point
  - Properly returns 0: 1 point
  - Extraneous base cases: no penalty if they don’t cause problems, 1 point penalty if they do

- Recursion: 13 points
  - Properly assumes return value will be `INT_MAX` unless some merge sequence is recursively discovered: 1 point
  - Properly iterates over all separation points, treating it either as the separator between the first two sets to be merged (as I have in my solution) or the last two to be merged: 1 point
  - Bounds of the above iteration are correct: 1 point
  - Computes the cost to merge the two sets on either side of each partition: 1 point
  - Ignores partition if merge cost is `INT_MAX`: 1 point
  - If merge cost is finite, then properly manages recursion
    - Updates `Vector` as I have in preparation for recursive call: 1 point
    - (And does so correctly): 1 additional point
    - Correctly makes recursive call (as with my solution) or calls (with the other, unpublished solution) and catches return values: 1 point
    - If any recursive return values are `INT_MAX`, then ignore the rest of the separation: 1 point
    - If all return values are non-`INT_MAX`, then conditionally updates cost if a new minimum was discovered: 1 point
    - Knows to backtrack by restoring `Vector` to its pre-recursive call structure (or, in the case of the second approach, simply discards the temporaries): 1 point
    - (And backtracks correctly): 1 additional point (give them this point if no work was needed)
  - Returns min cost at the end: 1 point
Solution 3: Short Answer Questions [10 points, Median Grade: 6]

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.

   It makes sense to accept a copy if you know you need to make local changes in order to compute a result, all without changing the original.

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.

   • The Queue has a more constrained interface, so it optimizes its two dynamic enqueue and dequeue operations to run in constant time. The Vector is more general and can’t optimize `remove(0)`.

   • Algorithm narrative will almost certainly be better with words like enqueue and dequeue in place of phrases like add/append and remove(0).
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**?

Boggle would have still worked, but the computer search would have been much, much slower, since it would never prune.

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?

Recall that the memoization key involved the number of electoral votes needed from state i of 50 onward. If many of the states have the same number electoral votes, they are interchangeable from an electoral vote standpoint, there’s more degeneracy, and there are fewer-remaining-electoral-vote-counts-needed values contributing to cache keys.
e) [2 points] One of the sample calls I provided for Problem 2 was

```
minimumOpensNeeded({{1},{2},{1},{2},{4},{3},{3}}) 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.

Validation code could confirm that the input can be partitioned into one or more counting permutations, where a counting permutation is a permutation of the integers 1 through k, inclusive.