CS106B

Winter 2023

Practice Midterm Exam 2

This is a printable version of the second set of practice problems from the practice midterm exam. It's designed to have the same format and style as the actual CS106B midterm.

Solutions can be found by visiting the Additional Practice Problems resource on the CS106B website and looking up solutions to the corresponding problems.

Lexicon

```
Lexicon lex; Lexicon english(filename);
lex.addWord(word);

bool present = lex.contains(word);
bool pref = lex.containsPrefix(prefix);

int numElems = lex.size();
bool empty = lex.isEmpty();

lex.clear();

/* Elements visited in sorted order. */
for (string word: lex) { … }
```

Map

```
Map<K, V> map = {{k1, v1}, … {kn, vn}};

cout << map[key] << endl; // Autoinserts
map[key] = value; // Autoinserts

bool present = map.containsKey(key);
int numKeys = map.size();
bool empty = map.isEmpty();

map.remove(key);
map.clear();

Vector<K> keys = map.keys();
K key = map.firstKey();

/* Keys visited in ascending order. */
for (K key: map) { … }
```

Stack

```
stack.push(elem);

T val = stack.pop();   // Removes top
T val = stack.peek();  // Looks at top

int numElems = stack.size();
bool empty = stack.isEmpty();

stack.clear();
```

Queue

```
queue.enqueue(elem);

T val = queue.dequeue(); // Removes front
T val = queue.peek();   // Looks at front

int numElems = queue.size();
bool empty = queue.isEmpty();

queue.clear();
```

Set

```
Set<T> set = {v1, v2, …, vn};

set.add(elem); set += elem;

Set<T> s = set - elem; // or + elem

bool present = set.contains(elem);
set.remove(x); set -= x; set -= set2;

Set<T> unionSet = s1 + s2;
Set<T> intersectSet = s1 * s2;
Set<T> difference = s1 - s2;

T elem = set.first();

int numElems = set.size();
bool empty = set.isEmpty();

set.clear();

/* Visited in ascending order. */
for (T elem: set) { … }
```

Vector

```
Vector<T> vec = {v1, v2, …, vn};

vec[index]; // Read/write

vec.add(elem);  vec += elem;

vec.insert(index, elem);

vec.indexOf(elem); // index or -1

vec.remove(index);
vec.clear();

int numElems = vec.size();
bool empty = vec.isEmpty();

Vector<T> v = vec.subList(start, nElems);

/* Visited in sequential order. */
for (T elem: vec) { … }
```

string

```
str[index]; // Read/write

str.substr(start);
str.substr(start, numChars);

str.find(c); // index or string::npos
str.find(c, startIndex);

str += ch; str += otherStr;
str.erase(index, length);

/* Visited in sequential order. */
for (char ch: str) { … }
```

Grid

```
Grid<T> grid(nRows, nCols);
Grid<T> grid(nRows, nCols, fillValue);

int nRows = grid.numRows();
int nCols = grid.numCols();

if (grid.inBounds(row, col)) { … }

grid[row][col] = value;
cout << grid[row][col] << endl;

/* Visited left-to-right, top-to-bottom */
for (T elem: grid) { … }
```

Problem One: Rot13

Some online forums discuss topics where there's a potential for someone reading the post to have something "spoiled" for them. For example, when reading discussion of a current TV show, you might accidentally read a post that contains a major plot point before you've seen it.

There are many steps forums can take to make it less likely for this to happen. Some forums allow authors to mark a section as a spoiler, in which case a reader has to actively click on a "reveal" button to show the content. But not all forums do this, and so resourceful readers and writers have come up with a clever strategy for concealing spoilers: Rot13.

The idea behind Rot13 is simple. Essentially, you take the text that contains a spoiler, then replace each letter with the letter that appears 13 characters after it in the English alphabet. (If this would take you off the end of the alphabet, instead, wrap around and keep going.) For example, the letter A would be replaced by the letter N and the letter B would be replaced with the letter O. Similarly, the letter X would be replaced by the letter K (starting at X, we'd go Y, Z, A, B, C, D, E, F, G, H, I, J, K). Once you've Rot13'ed a piece of text, any spoilers it might contain won't be nearly as obvious. For example, the text

```
Fancr xvyyf Qhzoyrqber!
```

is a spoiler for a major piece of contemporary fiction. If you wanted to, you could work out what it says – but if you're content to leave it alone you can just ignore it.

The advantage of Rot13 is that if you apply Rot13 twice to the same piece of text, you'll get back what you started with. Therefore, hitting a piece of regular text with Rot13 will obfuscate it, and hitting that text with Rot13 will return it back to its original.

One downside of Rot13 is that it doesn't scramble numbers, punctuation, spaces, capitalization, etc. You can see from the above example that whatever it is I'm trying to say, it involves three words, of which the first and third are capitalized, and the last character is an exclamation point. But that's fine - the purpose of Rot13 is to avoid accidentally revealing something, not to encrypt it in a way that no one can read it. Gb yrnea ubj gb rapelcg grkg fb gung ab bar rkprcg gur vagraqrq erpvcvrag pna ernq vg, gnxr PF255, bhe pbhefr ba pelcgbtencul!

Write a function

```
string rot13(string input);
```

that returns the Rot13-ed version of a string. As a note, in C++, if you have a lower-case letter, you can determine its numeric index in the alphabet by writing `ch - 'a'`, and given a numeric index into the alphabet you can convert it to a character by writing `char(index + 'a')`. The same works for upper-case letters, just with `'A'` substituted for `'a'`.

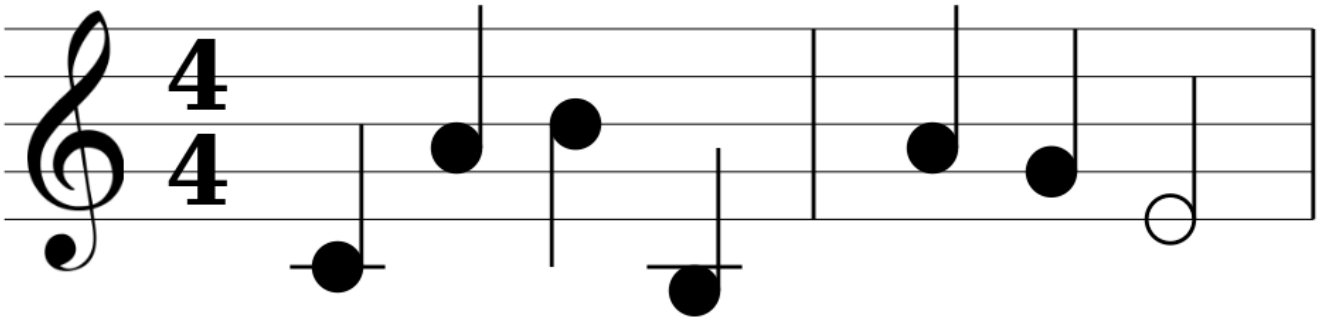
```
string rot13(string input) {
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work. There's additional space for your answer on the next page.

Additional space for your answer to Problem One, if you need it.

Problem Two: Musical Words

In Western musical notation, musical notes are denoted by the letters A, B, C, D, E, F, and G. Some English words, coincidentally, use only these letters. For example, the word cabbage is made up purely of these letters, and so you could, if you wanted to, play the word "cabbage" on an instrument.



A word like this is called a ***Western musical word***. Other words have this property as well. For example, the song "Badge" by Eric Clapton and George Harrison is often (incorrectly) said to take its name from the chord progression B-A-D-G-E, even though those aren't the notes used.

Write a function

```
Lexicon westernMusicalWordsIn(const Lexicon& english);
```

that takes as input a `Lexicon` of all words in English, then returns another `Lexicon` containing all the musical words in English. As a reminder, the `Lexicon` stores all its words in lowercase.

Similarly, in Indian classical music, the notes that comprise a scale are sa, re, ga, ma, pa, dha, and ni. Using these notes, our definition of a musical word changes, and we now get that words like *saga*, *papa*, and *mare* are musical.

Write a second function

```
Lexicon indianMusicalWordsIn(const Lexicon& english);
```

that works like your above function, but with Indian note names.

```
Lexicon westernMusicalWordsIn(const Lexicon& english) {
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work. There's additional space for your answer on the next page.

```
Lexicon indianMusicalWordsIn(const Lexicon& english) {
```

Additional space for your answer to Problem Two, if you need it.

Problem Three: Flightless Birds

Consider the following functions, `dodo` and `kiwi`, which differ only in the highlighted areas:

```
string dodo(const string& str, int n) {
  Queue<char> storage;

  /* Visit the characters in str in the
   * order in which they appear.
   */
  for (char ch: str) {
    storage.enqueue(ch);
  }

  for (int i = 0; i < n; i++) {
    char ch = storage.dequeue();
    storage.enqueue(ch);
  }

  /* If you don't initialize a string
   * in C++, it defaults to empty.
   */
  string result;
  while (!storage.isEmpty()) {
    result += storage.dequeue();
  }

  return result;
}
```

```
string kiwi(const string& str, int n) {
  Stack<char> storage;

  /* Visit the characters in str in the
   * order in which they appear.
   */
  for (char ch: str) {
    storage.push(ch);
  }

  for (int i = 0; i < n; i++) {
    char ch = storage.pop();
    storage.push(ch);
  }

  /* If you don't initialize a string
   * in C++, it defaults to empty.
   */
  string result;
  while (!storage.isEmpty()) {
    result += storage.pop();
  }

  return result;
}
```

Answer each of the following questions and fill in the appropriate blanks. No other justification is needed. Incorrect answers will not receive points, but otherwise there is no penalty for an incorrect guess.

i. **(1 Point)** How many strings `s` are there where `dodo(s, 3)` returns `"penguin"`?

- ☐ There are no choices of `s` for which this happens.
- ☐ There is exactly one choice of `s` where this happens, namely `s` = "_____".
- ☐ There are two or more choices of `s` where this happens, such as "_____" and "_____".

ii. **(1 Point)** How many strings `s` are there where `kiwi(s, 3)` returns `"penguin"`?

- ☐ There are no choices of `s` for which this happens.
- ☐ There is exactly one choice of `s` where this happens, namely `s` = "_____".
- ☐ There are two or more choices of `s` where this happens, such as "_____" and "_____".

iii. **(1 Point)** How many choices of `n` are there for which `dodo("kakapo", n)` returns `"kakapo"`?

- ☐ There are no choices of `n` for which this happens.
- ☐ There is exactly one choice of `n` for which this happens, namely `n` = _____.
- ☐ There are two or more choices of `n` for which this happens, such as _____ and _____.

iv. **(1 Point)** How many choices of `n` are there for which `kiwi("kakapo", n)` returns `"kakapo"`?

- ☐ There are no choices of `n` for which this happens.
- ☐ There is exactly one choice of `n` for which this happens, namely `n` = _____.
- ☐ There are two or more choices of `n` for which this happens, such as _____ and _____.

Problem Four: Complementary Strands

DNA strands are made up of smaller units called nucleotides. There are four different nucleotides present in DNA, which are represented by the letters A, C, G, and T. Any DNA strand, therefore, can be thought of as string consisting of the nucleotides that make up that strand, in order.

In DNA, each strand is paired with a complementary strand of the same length. The nucleotides in a complementary strand pair off as follows: A pairs with T, and C pairs with G. Here's an example of two complementary strands:

```
AACTTTGTACGTACGTACGTTGG
TTGAAACATGCATGCATGCAACC
```

Write a recursive function

```
bool areComplementaryStrands(const string& one, const string& two);
```

that takes as input two DNA strands, then returns whether those strands are complementary.

```
bool areComplementaryStrands(const string& one, const string& two) {
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work. There's additional space for your answer on the next page.

Additional space for your answer to Problem Four, if you need it.

Problem Five: Proofreading a Report

You've been working on preparing a report as part of a larger team. Each person on the team has been tasked with writing a different section of the report. To make sure that the final product looks good and is ready to go, your team has decided to have each person in the team proofread a section that they didn't themselves write.

There are a lot of ways to do this. For example, suppose your team has five members conveniently named A, B, C, D, and E. One option would be to have A read B's section, B read C's section, C read D's section,D read E's section, and E read A's section, with everyone proofreading in a big ring. Another option would be to have A and B each proofread the other's section, then have C proofread D's section, D read E's section, and E read C's section. A third option would be to have A read E's work, E read C's work, and C read A's work, then to have B and D proofread each other's work. The only restrictions are that (1) each section needs to be proofread by exactly one person and (2) no person is allowed to proofread their own work.

Write a function

```
void listAllProofreadingArrangements(const Set<string>& people);
```

that lists off all ways that everyone can be assigned a person's work to check so that no person is assigned to check their own work. For example, given the five people listed above, this function might print the following output:

A checks B, B checks A, C checks E, D checks C, E checks D

A checks B, B checks A, C checks D, D checks E, E checks C

A checks C, B checks A, C checks B, D checks E, E checks D

(... many, many lines skipped ...)

A checks C, B checks E, C checks D, D checks B, E checks A

A checks D, B checks C, C checks E, D checks B, E checks A

A checks E, B checks C, C checks D, D checks B, E checks A

Some notes on this problem:

- You're free to list the proofreading assignments in any order that you'd like. However, you should make sure that you don't list the same assignment twice.
- Your function should print all the arrangements it finds to `cout`. It shouldn't return anything.
- Your solution needs to be recursive – that's kinda what we're testing here.
- While in general you don't need to worry about efficiency, you should not implement this function by listing all possible permutations of the original group of people and then checking each one to see whether someone is assigned to themselves. That ends up being a bit too slow to be practical.
- Your output doesn't have to have the exact same format as ours. As long as you print out something that makes clear who's supposed to proofread what, you should be good to go. In case it helps, you may want to take advantage of the fact that you can use cout to directly print out a container class (`Vector`, `Set`, `Map`, `Lexicon`, etc.). For example, if you have a `Map` named `myMap`, the C++ code

```
cout << myMap << endl;
```

- prints out all the key/value pairs.
- As a hint, focus on any one person in the group. You know that they're going to have to proofread some section. Consider exploring each possible way they could do so.

```
void listAllProofreadingArrangements(const Set<string>& people) {
```

We will grade whatever you write in this box. Anything written outside of this region will be considered scratch work. There's additional space for your answer on the next page.

Additional space for your answer to Problem Five, if you need it.