CS106B Handout 17
Winter 2017 February 21, 2017

CS106B Midterm Exam

This exam is closed-book and closed-computer. You may have a double-sided, 8.5" × 11" sheet of notes with you when you take this exam. You may not have any other notes with you during the exam. You may not use any electronic devices (laptops, cell phones, etc.) during the course of this exam. Please write all of your solutions on this physical copy of the exam.

Unless otherwise indicated as part of the instructions for a specific problem, comments will not be required on the exam. Uncommented code that gets the job done will be sufficient for full credit on the problem. On the other hand, comments may help you to get partial credit if they help us determine what you were trying to do. You do not need to worry about efficiency unless a problem specifically requires an efficient solution.

This exam is "self-contained" in the sense that if you're expected to reference code from the lectures, textbook, assignments, or section handouts, we'll explicitly mention what that code is and provide sufficient context for you to use it. There's a reference sheet at the back of the exam detailing the library functions and classes we've discussed so far.

SUNetID: _____

Last Name: _____

First Name: _____

I accept both the letter and the spirit of the Honor Code. I have not received any unpermitted assistance on this test, nor will I give any. I do not have any advance knowledge of what questions will be asked on this exam. My answers are my own work. Finally, I understand that the Honor Code requires me to report any violations of the Honor Code that I witness during this exam.

(signed) _____

You have three hours to complete this exam. There are 40 total points.

Question	Points	Grader
(1) Container Classes	/ 8	
(2) Recursive Enumeration	/ 8	
(3) Recursive Optimization	/ 8	
(4) Recursive Backtracking	/ 8	
(5) Big-O and Efficiency	/ 8	
	/ 40	

<u>You can do this.</u> Best of luck on the exam!

Problem One: Container Classes (8 Points)

Skltns n r Clst ***(Recommended time spent: 20 minutes)***

The ***consonant skeleton*** of an English word is the string you get back when you start with that word and delete all the vowels from it. For example, the consonant skeleton of the word `skeleton` is `skltn`, the consonant skeleton of `exhilarating` is `xhlrtng`, and the consonant skeleton of `ukulele` is `kll`.

Many English words share the same consonant skeleton. For example, the words `oyster` and `austere` both share the consonant skeleton `str`, and the words `pasta` and `pesto` share the skeleton `pst`.

Your task is to write a function

```
string mostCommonConsonantSkeleton(const Lexicon& english);
```

that takes as input a `Lexicon` containing all the words in English, then returns the most commonly occurring consonant skeleton in the English language. For simplicity, you can assume we've provided you a function

```
bool isVowel(char ch);
```

that takes as input a character and returns whether or not it's a vowel.

Some notes on this problem:

- You can assume that all words are given in lower-case, so you shouldn't need to worry about case-sensitivity.
- Do not solve this problem by simply doing a double `for` loop over all possible pairs of English words. There are so many words that this approach is unlikely to finish running in any reasonable amount of time.
- If two or more strings are tied as the most common word skeleton, you can return any one of them.
- You can assume the input lexicon is nonempty.

```
/* Given a letter, returns true if it's a vowel and false otherwise. This function
 * is provided to you and you don't need to implement it.
 */
bool isVowel(char ch);


string mostCommonConsonantSkeleton(const Lexicon& english) {
```

(Extra space for your answer to Problem One, if you need it.)

Problem Two: Recursive Enumeration (8 Points)

Checking Your Work ***(Recommended time: 50 minutes)***

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`, you could write `cout << myMap << endl;` to print 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) {
```

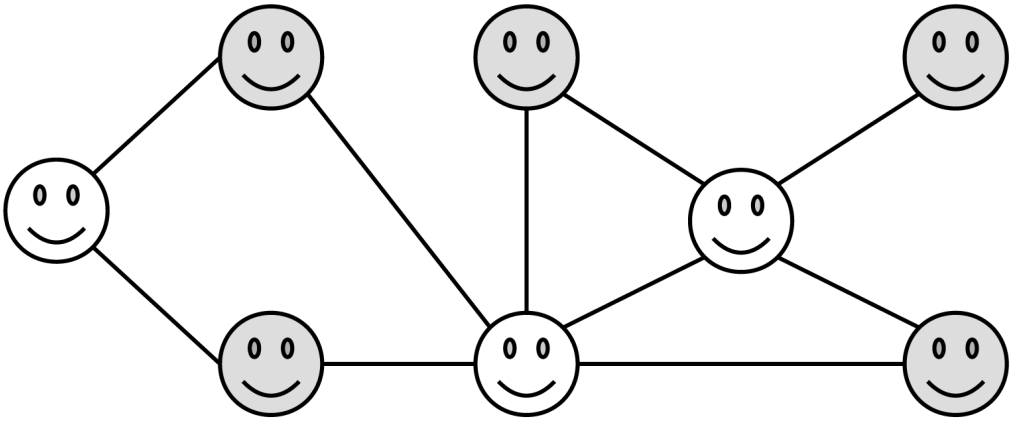
(Extra space for your answer to Problem Two, if you need it.)

Problem Three: Recursive Optimization (8 Points)

Avoiding Sampling Bias ***(Recommended time: 45 minutes)***

One of the risks that comes up when conducting field surveys is *sampling bias*, that you accidentally survey a bunch of people with similar backgrounds, tastes, and life experiences and therefore end up with a highly skewed view of what people think, like, and feel. There are a number of ways to try to control for this. One option that's viable given online social networks is to find a group of people of which no two are Facebook friends, then administer the survey to them. Since two people who are a part of some similar organization or group are likely to be Facebook friends, this way of sampling people ensures that you get a fairly wide distribution.

For example, in the social network shown below (with lines representing friendships), the folks shaded in gray would be an unbiased group, as no two of them are friends of one another.



Your task is to write a function

```
Set<string> largestUnbiasedGroupIn(const Map<string, Set<string>>& network);
```

that takes as input a `Map` representing Facebook friendships (described later) and returns the largest group of people you can survey, subject to the restriction that you can't survey any two people who are friends.

The `network` parameter is similar to the road network from the Disaster Preparation assignment. Each key is a person, and each person's associated value is the set of all the people they're Facebook friends with. You can assume that friendship is symmetric, so if person *A* is a friend of person *B*, then person *B* is a friend of person *A*. Similarly, you can assume that no one is friends with themselves.

Some other things to keep in mind:

- You need to use recursion to solve this problem – that's what we're testing here. ☺
- Your solution must not work by simply generating all possible groups of people and then checking at the end which ones are valid (i.e. whether no two people in the group are Facebook friends). Along the lines of the Disaster Preparation problem, this approach is far too slow to be practical.

```
Set<string> largestUnbiasedGroupIn(const Map<string, Set<string>>& network) {
```

(Extra space for your answer to Problem Three, if you need it.)

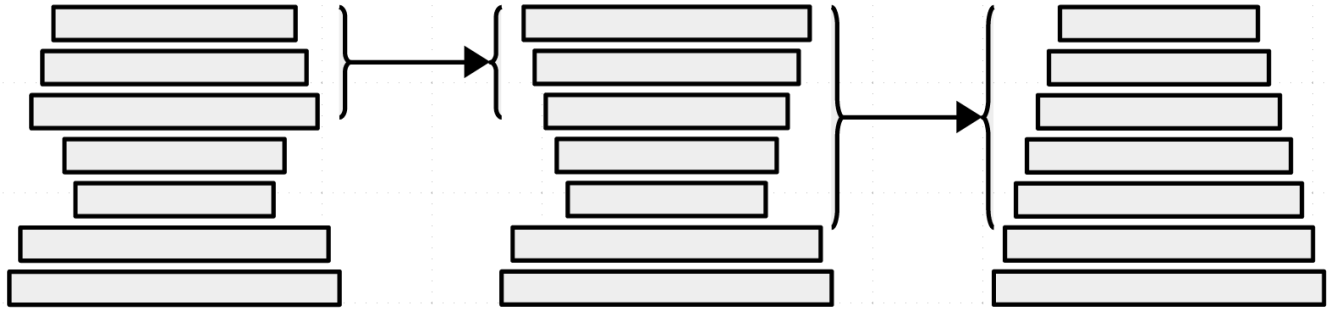
Problem Four: Recursive Backtracking (8 Points)

Mmmmm… Pancakes! ***(Recommended time: 45 minutes)***

Bill Gates is famous for a number of things – for his work at Microsoft, for his wide-scale philanthropy through the Bill and Melinda Gates Foundation, and, oddly enough, for a paper he coauthored about flipping pancakes.

Imagine you have a stack of pancakes of different diameters. In an ideal pancake stack, the kind of pancake stack you'd see someone photograph and put on Instagram, the pancakes would be stacked in decreasing order of size from bottom to top. That way, all the delicious syrup and melted butter you put on top of the pancakes can drip down onto the rest of the stack. Otherwise, you'll have a bigger pancake on top of a smaller one, serving as an edible umbrella that shields the lower pancakes from all the deliciousness you're trying to spread onto them.

The problem with trying to get a stack of pancakes in sorted order is that it's really, really hard to pull a single pancake out of a stack and move it somewhere else in the stack. On the other hand, it's actually *not* that difficult to slip a spatula under one of the pancakes, pick up that pancake and all the pancakes above it, then flip that part of the stack over. For example, in the illustration below, we can flip the top three pancakes in the stack over, then flip the top five pancakes of the resulting stack over:



Your task is to write a function

```
bool canSortStack(Stack<double> pancakes, int numFlips,
                  Vector<int>& flipsMade);
```

that accepts as input a `Stack<double>` representing the stack of pancakes (with each pancake represented by its width in cm) and a number of flips, then returns whether it's possible to get the stack of pancakes into sorted order making at most the specified number of flips. If so, your function should fill in the `flipsMade` outparameter with the sequence of flips that you made, with each flip specified as how many pancakes off the top of the stack you flipped. For example, the above illustration would correspond to the flip sequence {3, 5}, since we first flipped the top three pancakes, then the top five pancakes.

Here are some notes on this problem:

- Although this problem might seem similar to the Towers of Hanoi problem in that it involves a stack of items that end up in sorted order, ***this is a fundamentally different problem*** and you're not likely to make much progress with it if you approach it the same way. You really should treat this as a backtracking problem – try all possible series of flips you can make and see whether you can find one that gets everything into sorted order. As a result, don't worry about efficiency.
- Your solution must be recursive – again, that's what we're trying to test. ☺
- You can assume the pancakes all have different widths and that those widths are positive.
- The `Stack` here is ordered the way the pancakes are – the topmost pancake is at the top of the stack, and the bottommost pancake is at the bottom.
- You can assume `flipsMade` is empty when the function is first called, and its contents can be whatever you'd like them to be if you can't sort the pancake stack in the given number of flips.

```
bool canSortStack(Stack<double> pancakes, int numFlips, Vector<int>& flipsMade) {
```

(Extra space for your answer to Problem Four, if you need it.)

Problem Five: Big-O and Efficiency (8 Points)

Password Security ***(Recommended time: 20 minutes)***

Looking to change your password? Rather than picking a password that's a common word or phrase with a bunch of random numbers thrown in, consider using a multi-word password formed by choosing some sequence of totally random words out of the dictionary. Choosing four totally random words, it turns out, tends to be a pretty good way to make a password. Here's a couple passwords you might make that way:

```
RantingCollegersDenoteClinching
VivificationPandectYawnedCarmine
DetachednessHowlinglySportscastsVapored
UnlearnedMockeriesTuskedChuckles
SharpshootingPreyParaffinsLibeler
```

We generated these particular passwords using the following piece of code:

```
string makeRandomPassword(const Vector<string>& wordList) {
   string result;
   for (int i = 0; i < 4; i++) {
      int wordIndex = randomInteger(0, wordList.size() - 1);
      result += wordList[wordIndex];
   }
   return result;
}
```

When we ran this code, we used a word list containing about 120,000 words. There are other word lists available that we could have picked. The Basic English dictionary, for example, only has about 5,000 words. A more elaborate dictionary called ENABLE has about 180,000 words. It's therefore not all that unreasonable for us to analyze the efficiency of the above code in terms of the number of words *n* in our word list.

i. **(2 Points)** Let *n* denote the number of words in `wordList`. What is the big-O time complexity of the above code as a function of *n*? You can assume that the words are short enough that the cost of concatenating four strings together is O(1) and that a random number can be generated in time O(1). Explain how you arrived at your answer. Your answer should be no more than 50 words long.

ii. **(2 Points)** Let's suppose that the above code takes 1ms to generate a password when given a word list of length 50,000. Based on your analysis from part (i), how long do you think the above code will take to generate a password when given a word list of length 100,000? Explain how you arrived at your answer. Your answer should be no more than 50 words long.

Now, let's think about how someone might try to *break* your password. If they know that you chose four totally random English words, they could try logging in using every possible combination of four English words. Here's some code that tries to do that:

```
string breakPassword(const Vector<string>& wordList) {
  for (int i = 0; i < wordList.size(); i++) {
    for (int j = 0; j < wordList.size(); j++) {
      for (int k = 0; k < wordList.size(); k++) {
        for (int l = 0; l < wordList.size(); l++) {
          string password = wordList[i] + wordList[j] + wordList[k] + wordList[l];
          if (passwordIsCorrect(password)) {
            return password;
          }
        }
      }
    }
  }
}
```

As before, let's assume that the words in our word list are short enough that the cost of concatenating four of them together is O(1). Let's also assume that calling the `passwordIsCorrect` function with a given password takes time O(1).

iii. **(2 Points)** What is the *worst-case* big-O time complexity of the above piece of code as a function of *n*, the number of words in the word list? Explain how you arrived at your answer. Your answer should be no more than 50 words long.

iv. **(2 Points)** Imagine that in the worst case it takes 1,000 years to break a four-word password when given a word list of length 50,000. Based on your analysis from part (iii), how long will it take, in the worst case, to break a password when given a word list of length 100,000? Explain how you arrived at your answer. Your answer should be no more than 50 words long.

C++ Library Reference Sheet

Lexicon

```
Lexicon lex; Lexicon english(filename);
lex.addWord(word);
bool present = lex.contains(word);
bool pref = lex.containsPrefix(p);
int numElems = lex.size();
bool empty = lex.isEmpty();
lex.clear();
```

Map

```
Map<K, V> map = {{k1, v1}, … {kn, vn}};
map[key] = value; // Autoinsert
bool present = map.containsKey(key);
int numKeys = map.size();
bool empty = map.isEmpty();
map.remove(key);
map.clear();
Vector<K> keys = map.keys();
```

Stack

```
stack.push(elem);
T val = stack.pop();
T val = stack.top();
int numElems = stack.size();
bool empty = stack.isEmpty();
stack.clear();
```

Queue

```
queue.enqueue(elem);
T val = queue.dequeue();
T val = queue.peek();
int numElems = queue.size();
bool empty = queue.isEmpty();
queue.clear();
```

Set

```
Set<T> set = {v1, v2, …, vn};
set.add(elem);
set += 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();
```

Vector

```
Vector<T> vec = {v1, v2, …, vn};
vec.add(elem);
vec += elem;
vec.insert(index, elem);
vec.remove(index);
vec.clear();
vec[index]; // Read/write
int numElems = vec.size();
bool empty = vec.isEmpty();
vec.subList(start, numElems);
```

TokenScanner

```
TokenScanner scanner(source);
while (scanner.hasMoreTokens()) {
   string token = scanner.nextToken();
   …
}
scanner.addWordCharacters(chars);
```

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);
```

ifstream

```
input.open(filename);
input >> val;
getline(input, line);
```

GWindow

```
GWindow window(width, height);
gw.drawLine(x0, y0, x1, y1);
pt = gw.drawPolarLine(x, y, r, theta);
```

GPoint

```
double x = pt.getX();
double y = pt.getY();
```

General Utility Functions

```
int getInteger(optional-prompt);
double getReal(optional-prompt);
string getLine(optional-prompt);
int randomInteger(lowInclusive,
                  highInclusive);
double randomReal(lowInclusive,
                  highExclusive);
error(message);
x = max(val1, val2); y = min(val1, val2);
stringToInteger(str); stringToReal(str);
integerToString(intVal);
realToString(realVal);
```