CS106B Handout 17A

Winter 2017 February 21, 2017

Alternate 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. ***I will not discuss this exam with <u>anybody</u> until 10PM on Tuesday, February 21st.***

(signed) _____

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

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

You can do this. Best of luck on the exam!

Problem One: Container Classes (8 Points)

A Matter of Context ***(Recommended time spent: 25 minutes)***

A recent issue of the New York Times profiled Google's new translation system. It's powered by a technique called ***word2vec*** that builds an understanding of a language by looking at the context in which each word occurs.

Imagine you have a word like "well." There are certain words that might reasonably appear immediately before the word "well" in a sentence, like "feeling," "going," "reads," etc., and some words that that are highly unlikely to appear before "well," like "cake," "circumspection," and "election." The idea behind word2vec is to find connections between words by looking for pairs of words that have similar sets of words preceding them. Those words likely have some kind of connection between them, and the rest of the logic in word2vec works by trying to discover what those connections are.

Your task is to write a function

```
Map<string, Lexicon> predecessorMap(istream& input);
```

that takes as input an `istream&` containing the contents of a file, then returns a `Map<string, Lexicon>` that associates each word in the file with all the words that appeared directly before that word. For example, given JFK's quote

"Ask not what your country can do for you; ask what you can do for your country,"

your function should return a Map with these key/value pairs:

```
   "not" : { "ask" }
  "what" : { "not", "ask" }
  "your" : { "what", "for" }
"country" : { "your" }
   "can" : { "country", "you" }
    "do" : { "can" }
   "for" : { "do" }
   "you" : { "for", "what" }
   "ask" : { "you" }
```

Notice that although the word "ask" appears twice in the quote, the first time it appears it's the first word in the file and so nothing precedes The second time it appears, it's preceded by some whitespace and a semicolon, but before that is the word "you," which is what ultimately appears in the `Lexicon`.

Some notes on this problem:

- You can assume that a token counts as a word if its first character is a letter. You can use the `isalpha` function to check if a character is a letter.
- Your code should be case-insensitive, so it should return the same result regardless of the capitalization of the words in the file. The capitalization of the keys in the map is completely up to you.
- Your code should completely ignore non-word tokens (whitespace, punctuation, quotation marks, etc.) and just look at the words it encounters.
- It is not guaranteed that the file has any words in it, and there's no upper bound on how big the file can be.
- You'll almost certainly want to use `TokenScanner` here. Don't worry about setting up your `TokenScanner` so that it skips over whitespace or treats single quotes as letters – you can do that if you'd like, but you've already proven in Assignment 1 that you know how to do that.

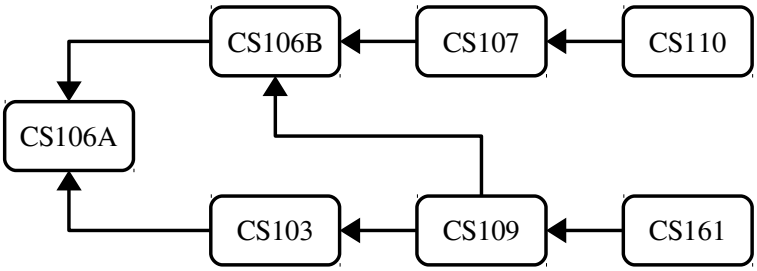
```
Map<string, Lexicon> predecessorMap(istream& input) {
```

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

Problem Two: Recursive Enumeration (8 Points)

Task Planning ***(Recommended time: 45 minutes)***

Imagine you have some collection of tasks that need to be done. Some of those tasks might depend on one another. For example, you might be navigating the CS Core, shown here:



Here, the arrows indicate prerequisites. CS106B has CS106A as a prerequisite, CS110 has CS107 as a prerequisite, CS109 has both CS106B and CS103 as prerequisites, and CS106A has no prerequisites. Assuming you can only take one CS class per quarter, what possible orderings are there for these classes that don't violate any prerequisites? Your task is to write a function

```
void listLegalOrderingsOf(const Map<string, Set<string>>& prereqs);
```

that takes as input a `Map` representing the prerequisite structure, then lists all possible orders in which you could complete those tasks without violating the prerequisites. The `prereqs` map is structured so that each key is a task and each value is the set of that task's immediate prerequisites. For example, the CS Core would be represented by the following map:

```
 "CS103" : { "CS106A" }
"CS106A" : { }
"CS106B" : { "CS106A" }
 "CS107" : { "CS106B" }
 "CS109" : { "CS103", "CS106B" }
 "CS110" : { "CS107" }
 "CS161" : { "CS109" }
```

Given this prerequisite structure, your function would then print out all of the following:

```
CS106A, CS106B, CS107, CS110, CS103, CS109, CS161
CS106A, CS103, CS106B, CS109, CS161, CS107, CS110
CS106A, CS106B, CS107, CS103, CS109, CS161, CS110
              (… many, many more …)
```

Some notes on this problem:

- Every task will be present in the `Map`. A task with no prerequisites will be represented as a key whose value is an empty `Lexicon`, as is the case for CS106A in the above example.
- You can assume that the tasks actually can be legally ordered and that there won't be any weird cases where a set of tasks all mutually depend on one another.
- Your function must not list off the same ordering twice.
- Your function must not work by simply generating all possible permutations of the tasks and then printing out just the ones that obey all the constraints. Along the lines of the Disaster Preparation problem, that would just be too inefficient.
- Your solution must be recursive, since that's kinda what we're trying to test here. ☺
- Your output doesn't have to exactly match our format. List off the orderings in whatever format you'd like. In case it helps, you can directly print a `Map`, `Set`, `Vector`, or `Lexicon` to `cout`.

```
void listLegalOrderingsOf(const Map<string, Set<string>>& prereqs) {
```

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

Problem Three: Recursive Optimization (8 Points)

Taking Care of Business ***(Recommended time: 45 minutes)***

You are working on a team project with a number of other folks. That project has a number of tasks that need to get completed, and fortunately they're all independent of one another. The project is completed as soon as all the tasks have been done, and you're interested in finding the fastest way to accomplish this. For example, if one person ends up working for 16 hours, another for 2 hours, and a third for 5 hours, it'll take 16 hours for the project to be completed, since you'll have to wait for the person who has 16 hours' worth of tasks to finish their work. On the other hand, if one person ends up working for 7 hours, another for 8 hours, and the third for 7 hours, then the project will be completed in 8 hours, since that's the length of time the longest person needs to complete their tasks.

Imagine that each task is represented as this handy `struct`:

```
struct Task {
      string name;     // The name of the task
      int hoursNeeded; // How long it takes to complete.
};
```

Write a function

```
Map<string, Vector<Task>> fastestAllocationOf(const Set<string>& people,
                                              const Vector<Task>& tasks);
```

that takes as input a set containing the names of all the people on the team along with the list of all the tasks to complete, then returns a `Map<string, Vector<Task>>` assigning people (by name) to the list of tasks they need to complete. The particular schedule you return should be chosen so that the very last person in that schedule to finish finishes as early as possible.

Some notes on this problem:

- There will always be at least one person in the group.
- If there are multiple schedules that all have equally good completion times, you can choose any one of them to return.
- Tasks never take a negative amount of time to complete.
- Your solution needs to use recursion – that's kinda what we're testing here. ☺
- If someone is assigned no tasks, you can either include their name as a key in the map associated with an empty `Vector`, or you return a map that doesn't associate their name with anything, whichever you find easier.

```
struct Task {
      string name;     // The name of the task
      int hoursNeeded; // How long it takes to complete.
};


Map<string, Vector<Task>> fastestAllocationOf(const Set<string>& people,
                                              const Vector<Task>& tasks) {
```

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

Problem Four: Recursive Backtracking (8 Points)

Waste Not, Want Not ***(Recommended time: 45 minutes)***

In a recent study, the USDA Economic Research Service determined that roughly 30% of all the food grown in the United States is wasted. That's over $150 billion in wasted food, so much in fact that wasted food in landfills is the third biggest source of carbon emissions in the United States. Being the responsible person that you are, and being a recursion veteran, you decide to see if you can find a way to plan your own meals so that you waste as little food as possible.

Let's imagine that you have your pantry represented as a `Map<string, int>`, where the key is the name of whatever ingredient is in your pantry (`"black beans"`, `"potatoes"`, `"barberries"`, etc.) and the value is how much of that ingredient you have lying around (in some arbitrary choice of units).

You also have a list of recipes you can make. Each recipe is represented as a struct:

```
struct Recipe {
    string name;
    Map<string, int> ingredients;
};
```

Here, the ingredients are *also* represented as a `Map<string, int>` mapping ingredients to how many copies of that ingredient you need for the recipe. For example, the delicious Persian rice dish adas polow, which needs a lot of rice, some lentils, and a little saffron, might look like this:

```
{{"rice", 4}, {"lentils", 2}, {"saffron", 1}}
```

A recipe for ful medames, the national dish of Sudan, might have these ingredients:

```
{{"fava beans", 3}, {"oil", 2}, {"cumin", 1}}
```

You're interested in planning out your meals for the week and want to see whether it's possible to make some combination of dishes that collectively uses up everything in your pantry. To do so, your job is to write a function

```
bool canGetZeroWaste(const Map<string, int>& pantry,
                     const Vector<Recipe>& recipes,
                     Vector<Recipe>& foodPlan);
```

that takes as input your pantry contents and a list of all the recipes you know how to make, then returns whether it's possible to find list of recipes to make that collectively use up everything in your pantry. If so, you should fill in the `foodPlan` argument with a list of all the meals you would make.

As an important note on this problem, ***you should account for the case where you choose to make multiple copies of the same recipe***. For example, you may find that the best way to exhaust your pantry would be to make ten batches of adas polow and three batches of ful medames. If you do make multiple copies of the same recipe, you should include an appropriate number of copies in the `foodPlan` outparameter.

You can assume that `foodPlan` is empty when the function is first called and the contents of `foodPlan` are irrelevant if your function returns false. You can also assume you have access to a function

```
bool haveIngredientsFor(const Recipe& r, const Map<string, int>& pantry);
```

that takes as input a recipe and the contents of a pantry, then returns whether the pantry has the ingredients necessary to make the recipe.

Oh, and you actually need to use recursion here. ☺

```
struct Recipe {
    string name;
    Map<string, int> ingredients;
};


/* Given a recipe and the contents of a pantry, returns whether there are
 * enough ingredients in the pantry to make the recipe. You do not need to
 * implement this function.
 */
bool haveIngredientsFor(const Recipe& r, const Map<string, int>& pantry);


bool canGetZeroWaste(const Map<string, int>& pantry,
                     const Vector<Recipe>& recipes,
                     Vector<Recipe>& foodPlan) {
```

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

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

Some Sort of Sort ***(Recommended time: 20 minutes)***

Let's suppose that you are working with some code that uses a mystery sorting algorithm. You run that sorting algorithm on three different classes of inputs: inputs that are already in sorted order, inputs that are in reverse-sorted order, and inputs where the elements are in a totally random order. You measure the amount of time it takes for the sorting algorithm to complete on different inputs of this sort and get back the following data table:

Input Size	Sorted Order	Random Order	Reverse Order
10,000	271μs	0.39s	0.78s
20,000	428μs	1.53s	3.08s
30,000	614μs	3.47s	7.05s
40,000	760μs	6.13s	12.3s
50,000	876μs	9.54s	19.4s
60,000	1030μs	13.7s	28.0s
70,000	1130μs	18.8s	38.0s
80,000	1350μs	24.4s	51.9s

This question explores what you can infer from this data. Note that the data in the "Sorted Order" column are in *microseconds* and the data in the other two columns are in *seconds*.

i. **(3 Points)** Based on the data available to you, what is your best guess about the big-O runtime of this sorting algorithm on an input consisting of *n* elements in *sorted* order? Justify your answer in at most fifty words.

ii. **(3 Points)** Based on the data available to you, what is your best guess about the big-O runtime of this sorting algorithm on an input consisting of *n* elements in *reverse-sorted* order? Justify your answer in at most fifty words.

iii. **(2 Points)** Based on the data available to you, is this algorithm most likely selection sort, insertion sort, mergesort, or something else? Justify your answer in at most fifty words.

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);
```