Alternate CS106B Midterm Exam (v2)

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 anybody 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	

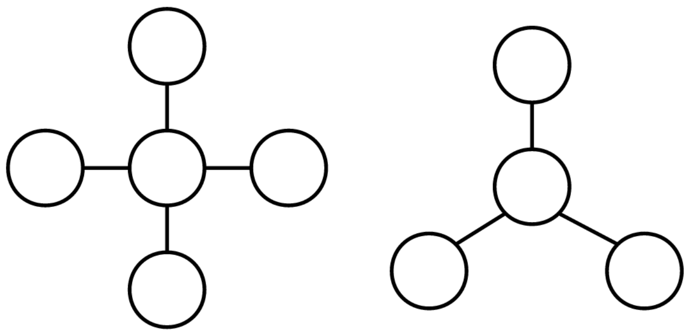
You can do this. Best of luck on the exam!

Problem One: Container Classes (8 Points)

No More Counting Dollars, We'll Be... ***(Recommended time spent: 25 minutes)***

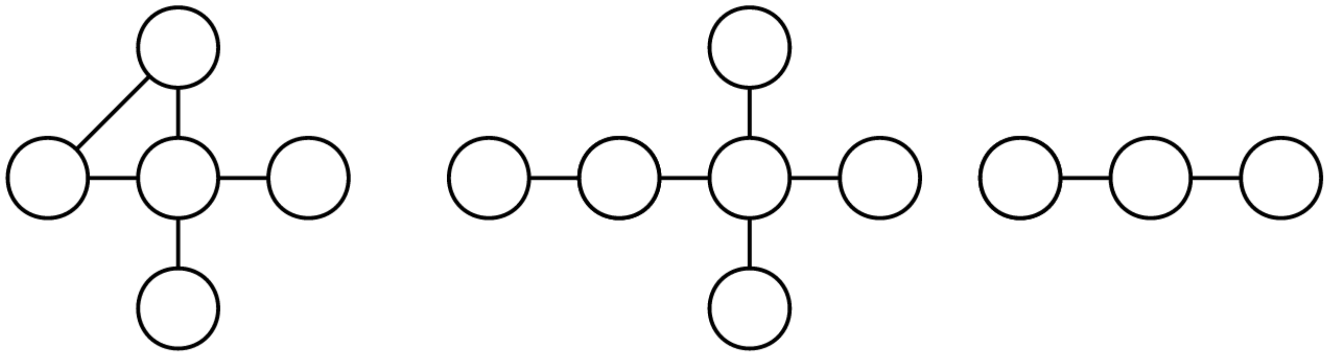
Imagine you have a road network for a country represented as a `Map<string, Set<string>>`, as in the Disaster Planning assignment. Each key is the name of a city, and the associated value consists of all the cities that are adjacent to that city. You can assume that the road network is bidirectional: if *A* is adjacent to *B*, then *B* is adjacent to *A*. You can also assume no city is adjacent to itself.

A ***star*** is a cluster of four or more cities arranged as follows: there's a single city in the center that's connected to all the other cities in the cluster, and every other city in the cluster is only connected to the central city. Here are some stars shown below:



Stars commonly arise in road networks in smaller island nations: you have a main, central city (often a capital city) and a bunch of smaller, outlying towns.

Here are some examples of groups of cities that aren't stars. The group on the left isn't a star because two of the peripheral cities are connected to one another (and therefore not just the central city). The group in the center isn't a star because one of the peripheral cities is connected to a city besides the central one. Finally, the group on the right isn't a star because it doesn't have the minimum required number of nodes.



If you have a country that consists of a large archipelago, you might find that its road network consists of multiple different independent stars. In fact, the number of stars in a road network is a rough proxy for how decentralized that country is.

Your task is to write a function

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

that takes as input the road network, then returns the number of stars in the network. You don't need to worry about efficiency, but do be careful not to count the same star multiple times.

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

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

Problem Two: Recursive Enumeration (8 Points)

Onward and Upward ***(Recommended time: 45 minutes)***

One of the warm-up problems we gave you on Assignment 3 was to test whether a particular string was a subsequence of another string. As a refresher, if *S* and *T* are strings, we say that *S* is a subsequence of *T* if all of the letters of *S* appear in *T* in the same relative order that they appear in *S*. For example, `pin` is a subsequence of the **p**rogramm**in**g, and the string `singe` is a subsequence of **s**pr**ing**tim**e**. However, `steal` is not a subsequence of `least`, since the letters are in the wrong order, and `i` is not a subsequence of `team` because there is no `i` in `team`. The empty string is a subsequence of every string, since all 0 characters of the empty string appear in the same relative order in any arbitrary string.

We can generalize this idea to talk about subsequences not just of strings, but of arbitrary lists of objects. For example, we could say that (1, 3, 7) is a subsequence of (**1**, 2, **3**, 4, 5, 6, **7**) and that (28, 31, 30, 10) is a subsequence of (34, **28**, 41, **31**, **30**, 22, **10**, 23, 16).

An ***increasing subsequence*** of a list of numbers is a subsequence of that list whose elements are in sorted order. For example, the sequence 106, 107, 110 is an increasing subsequence of the sequence (**106**, 103, 109, **107**, 161, **110**), and the sequence (11, 17, 26, 40) is an increasing subsequence of this longer sequence:

(22, **11**, 34, **17**, 52, **26**, 13, **40**, 20, 10, 5, 16, 8, 4, 2, 1)

Note that any zero-element or one-element sequence counts as an increasing subsequence.

Your task is to write a recursive function

```
void listIncreasingSubsequencesOf(const Vector<int>& values);
```

that takes as input a `Vector<int>` containing a list of numbers, then prints out every increasing subsequence of the original list.

Some notes on this problem:

- The order in which you list the different increasing subsequences is irrelevant.
- You can assume there are no duplicate elements in the `Vector`.
- Make sure that you don't list the same subsequence twice.
- You can directly print out a `Vector<int>` to `cout`. For example:

  ```
  Vector<int> myVec;
  cout << myVec << endl; // Totally legal!
  ```
- You should *not* solve this problem by generating all possible subsequences of the original sequence, then only printing the ones that happen to be ascending. This is too inefficient.
- You need to use recursion to solve this problem – that's what we're testing here. ☺

```
void listIncreasingSubsequencesOf(const Vector<int>& values) {
```

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

Problem Three: Recursive Optimization (8 Points)

Triangle Pathfinding ***(Recommended time: 40 minutes)***

Imagine you have a triangle of numbers, like the one shown here:

```
                137
            42     271
        314    613    53
    650    408    106    519
281    159    812   265    459
```

You're interested in finding a path from the top of the numbers down to the bottom whose total sum is as low as possible. The rule is that at each step you can only move down and to the right one step or down and to the left one step, like we did in Pascal's triangle or the Human Pyramids problem. The optimal path down this particular triangle is shown in bold and sums to 832.

We can represent a path down a triangle using the following `struct`s. This first `struct` represents the location of a number in the triangle:

```
struct Location {
    int row;
    int col;
};
```

The grid uses the same coordinate system as in the Human Pyramids problem. Row 0 is the topmost row, Row 1 the row below that, Row 2 the row below that, etc. Column 0 within any row is the leftmost number, Column 1 is the number second to the left, Column 2 is the number third from the left, etc.

Our second `struct` represents a path down the triangle:

```
struct PathInfo {
    int pathCost;
    Vector<Location> sequence;
};
```

Here, `pathCost` is the sum of all the numbers on the path, and `sequence` is the sequence of steps taken through the triangle as part of that path. Your task is to write a function

```
PathInfo bestPathThrough(const Grid<int>& numbers);
```

that takes as input a grid of numbers (described below) and returns a `PathInfo` corresponding to the lowest-cost path from the top of the triangle to the bottom. To receive full credit for your solution, you ***must*** use memoization in the course of solving this problem. This problem is very similar in structure to the Human Pyramids problem, so we hope this doesn't cause too much trouble.

Some notes on this problem:

- The optimal path starting at a particular point in the triangle is ***not*** always the one you'd take by moving to the smaller of the two numbers below you. In the above triangle, for example, the first step of the path passes up on the 42 and instead opts for the 271, even though the 42 is a smaller number. The reverse is true as well – if you walk up the triangle, it's not always ideal to move upward to the smaller of the two numbers above you.
- You can access an individual location in a grid by writing *grid*[*row*][*col*]. You can get the number of rows by writing *grid*`.numRows()` and the number of columns by writing *grid*`.numCols()`.
- You must use recursion when solving this problem, since that's ultimately what we're interested in testing here. ☺

```
struct Location {
    int row;
    int col;
};

struct PathInfo {
    int pathCost;
    Vector<Location> sequence;
};

PathInfo bestPathThrough(const Grid<int>& numbers) {
```

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

Problem Four: Recursive Backtracking (8 Points)

Tug-of-War ***(Recommended time: 45 minutes)***

You're interested in organizing a dorm tug-of-war contest. To keep things as fair as possible, you want to split everyone in your dorm into two teams of roughly equal strength.

Imagine you have a `struct`, like this one, representing a person:

```
struct Person {
    string name;
    int pullingPower;
};
```

Your task is to write a function

```
bool canDivideEvenly(const Vector<Person>& people,
                     Set<string>& teamOne, Set<string>& teamTwo);
```

that takes as input a list of people, then returns whether there's a way to divide them into two groups whose combined pulling power is ***exactly*** equal. If so, you should fill in the `teamOne` and `teamTwo` outparameters with the names of the people on each team. For example, consider this list of people:

Anabolic Aashna: Pulling Power 1,400
Steroid Steve: Pulling Power 1,000
Buff Belinda: Pulling Power 800
Fit Fatima: Pulling Power 600
Mediocre Malcolm: Pulling Power 500
Average Ahmed: Pulling Power 500
Lazy Larry: Pulling Power 400
Scrawny Sylvester: Pulling Power 200

In this case, it is possible to split the group into two teams: put Aashna, Belinda, and Malcolm on one team (combined pulling power 2,700) and Steve, Fatima, Ahmed, Larry, and Sylvester on the other (combined pulling power 2,700).

Some notes on this problem:

- If there are many different ways for the teams to be split evenly in half, you can choose any one of them.
- You can assume that the `teamOne` and `teamTwo` parameters are empty when the function is initially called, and their contents can be arbitrary if your function returns false.
- Your function must operate recursively, since, as the name of this problem suggests, that's kinda what we're trying to test. ☺

```
struct Person {
    string name;
    int pullingPower;
};


bool canDivideEvenly(const Vector<Person>& people,
                     Set<string>& teamOne, Set<string>& teamTwo) {
```

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

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

The Slower Option ***(Recommended time: 20 minutes)***

If you know the big-O runtime of a piece of code, you can make reasonably good estimates of how long it will take to complete on an input of unknown size based on what you've seen of its runtime on inputs of other sizes.

Some runtimes, like O(n) and O(n^2), scale quite well. Other runtimes scale quite poorly, and this question explores just how poorly those runtimes scale.

i. **(3 Points)** Let's imagine that you have an algorithm whose runtime is O(2^n). You've run the code on an input of size 100 and (somehow) measured that it takes 1,000 years to complete. Based on the big-O runtime of the algorithm, how long do you anticipate it will take the code to finish when run on an input of size 101? Justify your answer in fifty words or fewer.

ii. **(3 Points)** Let's imagine that you have an algorithm whose runtime is O(n!). You've run the code on an input of size 100 and (somehow) measured that it takes 1,000 years to complete. Based on the big-O runtime of the algorithm, how long do you anticipate it will take the code to finish when run on an input of size 101? Justify your answer in fifty words or fewer.

iii. **(2 Points)** Let's imagine that you have an algorithm whose runtime is O(n^6). You've run the code on an input of size 100 and (somehow) measured that it takes 1,000 years to complete. Based on the big-O runtime of the algorithm, how long do you anticipate it will take the code to finish when run on an input of size 200? Justify your answer in fifty words or fewer.

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 = {{k_1, v_1}, … {k_n, v_n}};
  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 = {v_1, v_2, …, v_n};
  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 = {v_1, v_2, …, v_n};
  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);
```