CS 106X, Autumn 2016
Midterm Exam

Your Name: _____

Section Leader: _____

Honor Code: *I hereby agree to follow both the letter and the spirit of the Stanford Honor Code. I have not received any assistance on this exam, nor will I give any. The answers I am submitting are my own work. I agree not to talk about the exam contents to anyone until a solution key is posted by the instructor.*

Signature: _____ ← **YOU MUST SIGN HERE!**

Rules: *(same as posted previously to class web site)*

- This exam is to be completed by each student **individually**, with no assistance from other students.
- You have **2 hours** (120 minutes) to complete this exam.
- This test is **open-book**, but **closed notes. You may not use any paper resources.**
- You may not use any computing devices, including calculators, cell phones, iPads, or music players.
- Unless otherwise indicated, your code will be graded on proper behavior/output, not on style.
- On code-writing problems, you do not need to write a complete program, nor `#include` statements. Write only the code (function, etc.) specified in the problem statement.
- Unless otherwise specified, you can write **helper functions** to implement the required behavior. When asked to write a function, do not declare any **global variables**.
- Do not abbreviate code, such as writing ditto marks ("") or dot-dot-dot marks (...).
- If you wrote your answer on a back page or attached paper, please **label this** clearly to the grader.
- Follow the Stanford **Honor Code** on this exam and correct/report anyone who does not do so.

Good luck! You can do it!

Many thanks to colleagues for problem ideas:
Jerry Cain, Cynthia Lee, Nick Parlante, Chris Piech, Stuart Reges, Eric Roberts, Mehran Sahami, Keith Schwarz, and more.

#	Description	Earned	Max	Grader (initial)
1	Code Reading		9	
2	Code Reading		8	
3	Code Reading		8	
4	Code Writing		10	
5	Code Writing		10	
6	Code Writing		10	
7	Code Writing		10	
	TOTAL		**65**	

1. Code Reading

For both of the calls to the following recursive function below, indicate the console **output** that is printed, as well as what value is **returned**:

```
int mysteryX(int a, int b) {
    if (a < 10 || b < 10) {
        int c = a + b;
        cout << "c=" << c << " ";
        return c;
    } else if (a > b) {
        int x = mysteryX(b, a - b);
        int y = mysteryX(a / 2, b / 2);
        cout << "x=" << x << " y=" << y << " ";
        return x + y;
    } else {
        int z = mysteryX(a, b / 2);
        cout << "z=" << z << " ";
        return z;
    }
}
```

a) call:	`mysteryX(11, 15)`
output:	
returns:	

b) call:	`mysteryX(33, 49)`
output:	
returns:	

2. Code Reading

The following C++ code uses parameters and produces **five lines** of output. What is the output?
For the purposes of this problem, assume the following about the **memory addresses** in the program:

- `main`'s `a` variable is stored at address `0xaa00`
- `main`'s `b` variable is stored at address `0xbb00`
- `main`'s `c` variable is stored at address `0xcc00`
- `main`'s `d` variable is stored at address `0xdd00`
- `main`'s `e` variable is stored at address `0xee00`
- any memory dynamically allocated on the **heap** will be at address `0x777700`

```
int mysteryX(int* a, int b, int& c) {
    c++;
    b--;
    *a = (c + b);
    cout << c << " " << *a << " " << a << " " << b << endl;
    return b;
}

int main() {
    int a = 1;
    int b = 30;
    int c = 500;
    int* d = new int;
    *d = a;
    int* e = &b;

    mysteryX(d, c, b);
    b = mysteryX(&c, a, *e);
    mysteryX(e, c, a);

    cout << a << " " << b << " " << c << " " << d << " " << e << endl;
    cout << &a << " " << &b << " " << &c << " " << *d << " " << *e << endl;
    return 0;
}
```

3. Code Reading

Give a tight bound of the nearest runtime complexity class for each of the following code fragments in **Big-Oh notation**, in terms of variable *N*. (In other words, the algorithm's runtime growth rate as *N* grows.)Write a simple expression that gives only a power of *N*, not an exact calculation like $O(2N^3 + 4N + 14)$. Write your answer in the blanks on the right side.

Question	Answer

// a)
`Map<int, int> mapA;`
`for (int i = 1; i <= 2 * N; i++) {`
`  mapA.put(i, 42 + i);`
`}`
 	
`Set<int> setA;`
`for (int i = 1; i <= N; i++) {`
`  int value = mapA.get(i);`
`  setA.add(value);`
`  mapA.remove(i);`
`}`
`cout << "done!" << endl;`
	$O($_____$)$

// b)
`Vector<int> vecB;`
`for (int i = 1; i <= N; i++) {`
`  vecB.add(i);`
`}`
`Stack<int> stackB;`
`while (!vecB.isEmpty()) {`
`  stackB.push(vecB[vecB.size() - 1]);`
`  vecB.remove(vecB.size() - 1);`
`}`
`cout << "done!" << endl;`
	$O($_____$)$

// c)
`int sumC = 0;`
`for (int i = 0; i < 1000; i++) {`
`  for (int j = 1; j < N * 2; j++) {`
`  sumC++;`
`  }`
`  for (int j = 0; j < i; j++) {`
`  for (int k = 0; k < N * 5; k++) {`
`  sumC++;`
`  }`
`  }`
`}`
`cout << sumC << endl;`
	$O($_____$)$

// d)
`HashSet<int> setD;`
`for (int i = 0; i < 3 * N; i += 3) {`
`  setD.add(i);`
`  setD.add(i + 1);`
`  setD.add(i + 2);`
`}`
`Stack<int> stackD;`
`for (int i = 0; i < N * N; i++) {`
`  stackD.push(i);`
`}`
`for (int i = 0; i < N; i++) {`
`  setD.remove(i);`
`  stackD.pop();`
`  stackD.pop();`
`}`
`cout << "done!" << endl;`
	$O($_____$)$

4. Code Writing

Write a recursive function named **`combineQueues`** that accepts two queues of integers *Q1* and *Q2* by reference, whose contents are sorted, and returns a new queue *Q3* that contains all elements from both queues in **sorted** non-decreasing overall order, with *Q1* and *Q2* emptied. For example, if the queues store the following elements, with the front at left and the back at right:

```
Q1 = {1, 4, 4, 9, 15}                                   // state before call
Q2 = {2, 3, 4, 5, 6, 11, 28}
```

Then the call of `combineQueues(Q1, Q2)` would return the following queue of elements:

```
    {1, 2, 3, 4, 4, 4, 5, 6, 9, 11, 15, 28}             // return value
```

Note that the queues might not be the same size as each other initially. Either queue passed in might be empty. Remember that you can assume that the contents of *Q1* and *Q2* are in **sorted** non-decreasing order. The queues might contain duplicates, both internally and with respect to the other queue (as with the value 4 in our example call, which occurs twice in *Q1* and once in *Q2*).

Constraints: For full credit, obey the following restrictions. A solution that disobeys them can get partial credit.

- You need to create a result queue to return from your function. But outside of that, do not create or use any other auxiliary **data structures** like additional `Queue`s, `Stack`s, `Vector`, `Map`, `Set`, array, strings, etc.
- **Do not use any loops**; you must use recursion.
- Do not declare any **global variables**.
- Do not use any pre-existing library sorting functions in your solution, such as the STL `sort()` function.
- You *can* declare as many primitive variables like `int`s as you like.
- You *are* allowed to define other "**helper**" functions if you like; they are subject to these same constraints.

Write your answer on the next page.

4. Writing Space

(for grader only)	

5. Code Writing

Write a function named **`findBestFriends`** that accepts a reference to an input file of type **`ifstream`** representing a log of chat messages and prints the pair(s) of people who converse with each other most frequently.

The input file contains a collection of chat **messages**, one per line, in the format of the example shown at right. Each chat message begins with the sender's name, followed by a colon and a space, followed by the sender's message.

Chat messages are organized into **conversations**, which are groups of chat messages separated by a line of 3 dashes. For example, the file on the right contains 6 separate conversations.

Your function should process the input file and print out the names of the **pair**(s) of people who participated in the largest number of conversations together. If multiple pairs of people **tie** for the largest number of conversations, print all such pairs. It does not matter how many individual messages the people exchange, only how many conversations they both participate in together.

For example, if the input file stream for the file on the right is passed to your function, you should print the following output:

```
Bert and Ernie
Big Bird and Ernie
```

```
Ernie: hey bert!
Ernie: did you eat yet?
Bert: no, let's grab lunch!
Ernie: yay :)
---
Ernie: good morning!
Big Bird: Hello, there!
Ernie: have a great day
---
Ernie: bert, yt?
Ernie: guess not!
---
Bert: you see that cute dog?
Ernie: yeah I do! So cute
Ernie: I want a puppy now
---
C Monster: Who wants cookies?
Bert: Me!
Ernie: Me!
Big Bird: ooh I do!
Oscar: Not me!
C Monster: OK let's get some!
---
Big Bird: hey ernie!
Ernie: HEY! what's up?
---
```

The preceding is the correct output to print because both of those pairs participate in 3 conversations together. The pair of Bert and Ernie are in the first, fourth, and fifth conversations together; and the pair of Big Bird and Ernie are in the second, fifth, and sixth conversations together.

The relative order in which you print the pairs, and the order of the two individual names within each pair, does not matter. So printing "Ernie and Big Bird" or "Ernie and Bert" or any other ordering is also acceptable.

You may **assume valid file input**. Names are case sensitive, e.g. "bert" is a different name than "BERT" or "Bert."

Constraints: For full credit, obey the following restrictions. A solution that disobeys them can get partial credit.

- You should choose an **efficient** solution. Choose data structures intelligently and use them properly. While your code is not required to have any particular Big-Oh, you may lose points if your code is extremely inefficient.
- You may **read the file only once**. Do not re-open it or rewind the stream.
- You may use as many **collections** as you like, but you should choose them intelligently and use them properly.
- You may not define any custom classes or structs; you must use the Stanford collections to solve this problem.

Write your answer on the next page.

5. Writing Space

(for grader only)	

6. Code Writing

Write a recursive function named **`findSetCover`** that solves the problem described on this page. You are passed two parameters by reference: a set of integers *U* that we'll call the "universe", and a set-of-sets of integers *S* whose union is *U*. Your goal is to find and return an **"optimal" set-of-sets** made from sets in *S* whose union equals the universe. By "optimal" we mean that you should find the **fewest number of sets** from *S* that could be unioned together to produce *U*, breaking ties by the **fewest total number of elements** among the sets.

For example, suppose that our universe *U* and set-of-sets *S* have the following contents:

```
U = {1, 2, 3, 4, 5, 6}
S = {{}, {1}, {1, 2}, {1, 3}, {2, 3}, {2, 4, 6}, {3}, {3, 5, 6}, {4, 5}, {4, 6}, {6}}
```

If passed these sets, your function could **return** the following set-of-sets:

```
{{1, 2}, {3, 5, 6}, {4, 6}}                    // an optimal solution
```

This result contains 3 sets with a total of 7 elements. This is **optimal** because the union of its three sets is `{1, 2, 3, 4, 5, 6}`, which is our universe, and because there exists no way to generate such a union using fewer than three sets from *S*, nor any way to generate a union of *U* using three sets with a total of fewer than 7 elements. The following would be another **equally valid** optimal solution that your function could have returned:

```
{{1, 3}, {2, 4, 6}, {4, 5}}                    // another optimal solution
```

The following are some examples of sets-of-sets that would not be considered correct solutions to return. The first example is incorrect because the union of its sets is not *U* (it is missing the element 4). The second example is non-optimal because it uses four total sets, and a solution can be made using only three sets. The third example is non-optimal because its three sets have a total of 8 elements, and a solution can be made with three sets and only 7 elements.

```
{{1, 3}, {2, 3}, {3, 5, 6}}                    // invalid solution (union is not U)
{{1}, {2, 3}, {4, 5}, {6}}                     // non-optimal solution (four total sets)
{{1, 3}, {2, 4, 6}, {3, 5, 6}}                 // non-optimal solution (eight total elements)
```

You are guaranteed to be able to find a solution because you are to assume that the union of the sets in *S* is *U*, meaning that in a worst case your function could simply return the entirety of *S*.

Efficiency: While your code is not required to have any particular Big-Oh, your code should **prune its searching** and avoid examining solution paths that are guaranteed not to yield a correct solution to the problem. You may lose points if your code is extremely inefficient, visits unnecessary paths, repeatedly revisits the same path multiple times, or reexamines a given set from *S* more frequently than needed. You should also choose reasonable collections that efficiently implement the operations you need.

Constraints: For full credit, obey the following restrictions. A solution that disobeys them can get partial credit.

- Your function should not modify the collections that are passed to it; or, more accurately, if it does modify them, it must always return them to their original state before the function's work is complete.
- Do not declare any **global variables**.
- Your code can contain **loops** if needed to solve the problem, but to receive full credit, your overall algorithm must be recursive.
- You are allowed to define other **"helper"** functions if you like; they are subject to these same constraints.

Write your answer on the next page.

6. Writing Space

(for grader only)	

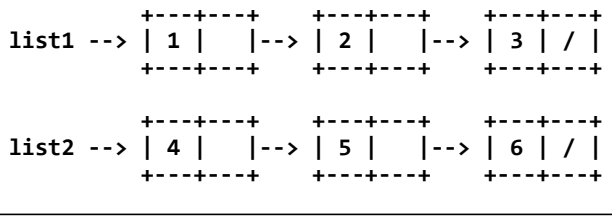
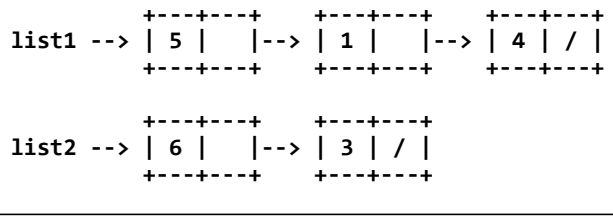
7. Code Writing

Write the code that will turn the "before" picture into the "after" picture by modifying links between the nodes shown.

If a given list node object does not appear in the "After" picture, you must **`delete` it** to free its memory.

Constraints: **You are *not* allowed to change any existing node's `data` field value.** Solve it by changing pointers.
You may declare **a single `ListNode*` pointer** variable (aside from `list1` and `list2`) to point to any existing node.
Do not construct any new `ListNode` objects (using the `new` keyword) in solving this problem.

To help maximize partial credit in case of mistakes, we suggest that you include optional **comments** with your code that describe the links you are trying to change, as shown in the solutions in our practice exams.

Before	After
	

Assume that you are using the `ListNode` structure as defined in lecture and section:

```
struct ListNode {
    int data;         // data stored in this node
    ListNode* next;   // a link to the next node in the list
    ...
};
```

(for grader only)	

Additional Writing Space