CS 106B Practice Midterm Exam #7

(based on CS 106B Autumn 2014 midterm)

This sample exam is intended to demonstrate an example of some of the kinds of problems that will be asked on the actual midterm exam. We do not guarantee that the number of questions in this sample exam will match the number on the real exam, nor that every kind of problem shown here will exactly match the kinds of problems shown on the final exam, though the real exam will be generally similar overall. Also, to save paper, this exam does not provide blank pages for writing space. The real exam will provide more blank space for you to write your answers.

In general, problem solutions are not graded on "style" as long as they work properly, though many questions do have some rules and constraints. For example, you should not use global variables, and some problems require recursion or a particular data structure.

1. Collections (read)

Write the **output** that would be produced by the following function when passed each of the following queues. Write the output exactly as it would appear on the console.

```
void collectionMystery7(Queue<int>& queue) {
    Stack<int> stack;
    int qsize = queue.size();
    for (int i = 0; i < qsize; i++) {
        if (queue.peek() % 2 == 0) {
            queue.enqueue(queue.dequeue());
        } else {
            stack.push(queue.peek());
            stack.push(queue.dequeue());
        }
    }
    while (!queue.isEmpty()) {
        stack.push(queue.dequeue());
    }
    while (!stack.isEmpty()) {
        cout << stack.pop() << " ";
    }
}
```

Note that the queues below are written in {front ... back} order.

---

a) `{1, 2, 3, 4, 5, 6}`

b) `{55, 33, 0, 88, 44, 99, 77, 66}`

c) `{80, 20, 65, 10, 5, 3, 40, 2, 11}`

2. Collections (write)

Write a function named **`coolest`** that accepts a string parameter storing a name of a file of information about Twitter followers and returns the name of the person who is the most "popular" in the data set, according to the following rules.

Twitter is a social network where users can "follow" each other to see each other's messages. Following a user is a one-directional relationship; if A follows B, it does not necessarily mean that B follows A. You might imagine that the most "popular" user is the one who has the most followers. But it is more impressive to be followed by people who have a lot of followers themselves. So we will define the "popularity" of a user to be the sum of the number of their followers' followers. For example, if user A has three followers named B, C, and D, then A's popularity is the sum of the number of followers that B, C, and D have. If B has 3 followers, C has 2, and D has 4, then A's popularity is 9. **(It does not matter if there is any overlap between the groups of people who follow B, C, and D in our example.)** One funny side effect of this popularity system is that you don't get any points for being followed by someone who has 0 followers.

In this problem, each line of the data file is in the format `"name1 name2"`, to indicate that the user *name1* follows the user *name2*. Your job is to read this data and return the name of the user with the highest popularity as described above. If two users tie for the highest popularity, return the one whose name comes earlier in alphabetical order.

For example, suppose a file named `twitter.txt` contains the lines below. Given this data set, your function would return `"Mehran"` because Mehran has the highest popularity score of 3 due to being followed by Stuart (who has 1 follower) and by Marty (who has 2 followers).

```
Stuart Marty
Helene Elmer
Donald Marty
Bruce Elmer
Donald Elmer
Donald Stuart
Stuart Mehran
Mehran Donald
Reid Elmer
Marty Mehran
```

You may **assume valid input**. Specifically, assume that the given file exists, is readable by your code, and that each line of it is in the format above with two whitespace-separated one-word names per line. Assume that the file contains at least one line of data. You may also assume that no user follows themselves and that there are no duplicate lines in the file. You do not need to worry about case-sensitivity on this problem; assume that each name always appears in the same case.

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

- You may open and read the contents of the file only once. Do not re-open it or rewind the stream.
- You may create up to **two additional data structures** (stack, queue, set, map, etc.) as auxiliary storage. A nested structure, such as a set of vectors, counts as one additional data structure. (You can have as many simple variables as you like, such as `int`s or `string`s.)
- Your solution should run in $O(N^2)$ time or faster, where N is the number of names in the file.

3. Big-Oh (read)

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. (Write the growth rate as N grows.) Write a simple expression that gives only a power of N, such as $O(N^2)$ or $O(\log N)$, *not* an exact calculation like $O(2N^3 + 4N + 14)$. Write your answer in the blanks on the right side.

Question	Answer

a)
`int sum = 0;`
`for (int i = 1; i <= N - 999; i++) {`
`  for (int j = 1; j <= 0.0001 * N/2; j++) {`
`  sum++;`
`  }`
`}`
`cout << sum << endl;`
	$O($_____$)$

b)
`int sum = 0;`
`for (int i = 1; i <= 1000000; i++) {`
`  sum++;`
`}`
`int x = 999999;`
`for (int i = 1; i <= x; i++) {`
`  for (int j = 1; j <= 999; j++) {`
`  sum++;`
`  }`
`  for (int k = 1; k <= 999; k++) {`
`  sum++;`
`  }`
`}`
`cout << sum << endl;`
	$O($_____$)$

c)
`Vector<int> vecC;`
`for (int i = 1; i <= N; i++) {`
`  vecC.insert(0, i);`
`}`
`Set<int> setC;`
`for (int i = 0; i < vecC.size(); i++) {`
`  setC.add(vecC[i]);`
`}`
`cout << "done!" << endl;`
	$O($_____$)$

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

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

4. Recursion (read)

For each call to the following recursive function, write the **output** that is produced as it would appear on the console. Recall that relational operators like < and > compare strings by alphabetical order; for example, "a" is less than "b".

```
void recursionMystery7(string s) {
    if (s.length() <= 1) {
        cout << s;
    } else {
        string first = s.substr(0, 1);
        string last  = s.substr(s.length() - 1, 1);
        string mid   = s.substr(1, s.length() - 2);
        if (first < last) {
            recursionMystery7(mid);
            cout << last << toUpperCase(first);
        } else {
            cout << "[" << first << "]";
            recursionMystery7(mid);
            cout << last;
        }
    }
}
```

Call	Output
a) recursionMystery7("abcd");	
b) recursionMystery7("leonard");	
c) recursionMystery7("breakfast");	

5. Recursion (write)

Write a recursive function named **`mergeDigitPairs`** that accepts an integer parameter *n* and returns the integer formed by combining each pair of digits from *n* into a single digit that is their sum. For example, if passed the number `1234`, you should combine the digits 12 into 1+2 or 3, and combine the digits 34 into 3+4 or 7, leading to a returned result of 37.

If adding a given pair of digits produces a two-digit number, repeat the process until you have a single-digit number to replace the original pair. For example, if passed the number `1168`, the 11 becomes 1+1 or 2, but the 68 becomes 6+8 or 14, so we merge them again by saying that 14 is 1+4 or 5, so the pair 68 turns into 5, leading to an overall result of 25.

If passed a number with an odd number of digits, the first (most significant) digit is left untouched. For example, the number 13372 becomes 169 because the 3+3 becomes 6 and the 7+2 becomes 9. If passed a negative number, perform the same process as usual but return a negative result. For example, when passed `-1234`, return -37. If passed a single-digit number, simply return that number itself.

The following table shows several calls and their expected return values:

Call	Return Value
`mergeDigitPairs(1234)`	`37`
`mergeDigitPairs(3186507)`	`3927`
`mergeDigitPairs(-52874)`	`-512`
`mergeDigitPairs(88888888)`	`7777`
`mergeDigitPairs(20581974)`	`2412`
`mergeDigitPairs(0)`	`0`
`mergeDigitPairs(6)`	`6`
`mergeDigitPairs(-14)`	`-5`

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

- Do not declare any **global variables**.
- **Do not use any loops**; you must use recursion.
- **Do not use a string** to solve this problem. (For example, do not convert *n* into a string.)
- Do not use any auxiliary data structures like `Vector`, `Map`, `Set`, array, etc.
  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.

6. Backtracking (write)

Write a recursive function named **`canPack`** that accepts information about a collection of bags of given sizes, and a collection of items of various sizes, and returns `true` if the given bags can hold all of the given items successfully. Use recursion and backtracking to try combinations of packing items into different bags, looking for a successful arrangement.

In this problem, a bag is a container (such as a backpack) that can store items inside of it. Bags and items are represented as integers: An integer for an item represents its size, and an integer for a bag represents the total size of items that can be stored inside it. For example, a bag of size 7 could store an item of size 7; or an item of size 5 and an item of size 2; or seven items of size 1; or any other combination up to a total size of 7 for the combined items inside it. A bag can be partially filled if necessary; a bag of size 7 could store items whose total weight is any number up to 7 inclusive.

Your function will accept **two parameters**. The first is a reference to a `Vector` of integers representing the sizes of the available bags. For example, the vector `{4, 8, 3, 3}` means that you have four bags available with sizes of 4, 8, 3, and 3 respectively. The second parameter to your function is a reference to a `Vector` of integers representing the available items. For example, the vector `{2, 3, 2, 5, 4, 1}` means that you have six items of sizes 2, 3, 2, 5, 4, and 1 respectively. Your function's task is to see if the given items can fit into the given bags in some combination. With this example vector of bags and of items, the function would return `true` because the items can be fit into the bags successfully. One successful arrangement is that the first bag of size 4 holds the size-4 item; the second bag of size 8 holds the size-3 and size-5 items; the third bag of size 3 holds a size-2 item, and the fourth bag of size 3 holds the other size-2 item and the size-1 item. The diagram below attempts to show this arrangement:

```
                                  [2,2]   [3,5]   [2]  [2,1]
bags:  {4, 8, 3, 3}          -->   {4,      8,     3,    3}   items in bags
items: {2, 3, 2, 5, 4, 1}
```

Below are several examples of combinations of bags and items where your function should return `false`, because there exists no combination where the items can be packed into the bags without overflowing.

`bags:  {12, 10}` `items: {8, 6, 7}`	`{5, 5, 5}` `{3, 3, 3, 3}`	`{1, 1, 5, 7, 9}` `{2, 4, 6, 8}`	`{}` `{1}`

If no items are passed (that is, if the collection of items passed is empty), your function should return `true`.

The collections passed to your function must be back to their original state at the end of the call. Either do not modify them, or if you modify them, fully undo your modifications before the function returns.

Hint: The point is not to devise a clever pattern or algorithms for finding the best packing strategy. The idea is to just try all possible choices of items in various bags until you find an arrangement that works.

For the most part you are not being graded on efficiency, but your code should not perform exactly the same unnecessary deep exploration multiple times. You should also avoid making copies of data structures extremely high numbers of times by always passing them by reference.

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

- Do not declare any **global variables**.
- Use any data structures you like, and your code can contain loops, but the overall algorithm must be recursive.
- You are allowed to define other **"helper"** functions if you like; they are subject to these same constraints.

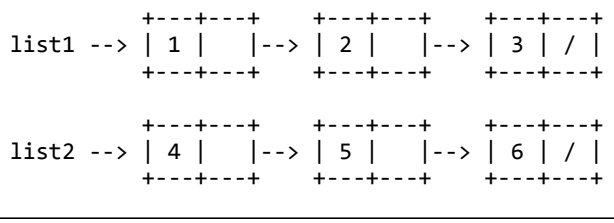
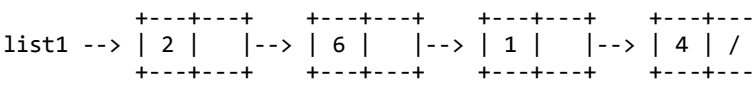
---

7. Pointers (write)

Write the code that will turn the "before" picture into the "after" picture by modifying links between the nodes shown. You are ***not*** allowed to change any existing node's `data` field value. You also should ***not*** construct new `ListNode` objects. You may declare **a single `ListNode*` pointer** variable (aside from `list1` and `list2`) to point to any existing node.

If a pointer variable does not appear in the "After" picture, it **doesn't matter** what value it has after the changes are made. If a given *node object* does not appear in the "After" picture, you must **free its memory** to avoid a memory leak.

To help maximize partial credit in case you make 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 and section handouts.

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
    ...
};
```
