CS 106B Practice Midterm Exam #2

(based on a CS 106B Autumn 2013 practice 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. C++ Basics and Parameters (read)

The following confusing C++ code uses parameters and produces four lines of output. What is the output?

```
string parameterMystery2(string& s1, string s2) {
    s1 += "1";
    s2 += "2";
    cout << s2 << " -- " << s1 << endl;
    return "!" + s2;
}

int main() {
    string a = "hi";
    string b = "bye";
    string c = "yo";

    parameterMystery2(a, c);
    parameterMystery2(c, b);
    string d = parameterMystery2(b, a);

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

2. File I/O and Strings (write)

Write a function named **`flipLines`** that accepts as its parameter a string holding a file name, opens that file and reads its contents as a sequence of lines, and writes to the console the same file's contents with successive pairs of lines reversed in order, with alternating capitalization. For example, if the input file named `carroll.txt` contains the following text:

```
TWAS brillig and the Slithy Toves
did GYRE and gimble in the Wabe.
All mimsey were the Borogroves,
and the mome RATHS outgrabe.

"Beware the Jabberwock, my Son,
the JAWS that bite, the claws that Catch,
Beware the JubJub bird and SHUN
The Frumious Bandersnatch."
```

Then the call of `flipLines("carroll.txt")` should print the first pair of lines in reverse order, then the second pair in reverse order, then the third pair in reverse order, and so on. It should produce the following console output:

```
DID GYRE AND GIMBLE IN THE WABE.
twas brillig and the slithy toves
AND THE MOME RATHS OUTGRABE.
all mimsey were the borogroves,
"BEWARE THE JABBERWOCK, MY SON,

BEWARE THE JUBJUB BIRD AND SHUN
the jaws that bite, the claws that catch,
THE FRUMIOUS BANDERSNATCH."
```

Notice the alternation between all-uppercase and all-lowercase. Also note that a line can be blank, as in the third pair. An input file can have an odd number of lines, as in the one above, in which case the last line is printed in its original position. You should not make any assumptions about how many lines are in the file.

If the input file does not exist or is not readable, your function should instead print the following output:

```
Unable to open input file "carroll.txt"!
```

---

3. ADTs / Collections (read)

Consider the following function:

```
void collectionMystery2(const Map<string, string>& m) {
    Set<string> s;
    for (string key : m) {
        if (m[key] != key) {
            s.add(m[key]);
        } else {
            s.remove(m[key]);
        }
    }
    cout << s << endl;
}
```

Write the output produced by the function when passed each of the following maps:

Vector	Output
a) `{"cast":"plaster", "house":"brick", "sheep":"wool", "wool":"wool"}`	
b) `{"ball":"blue", "corn":"yellow", "emerald":"green", "grass":"green", "winkie":"yellow"}`	
c) `{"apple":"peach", "corn":"apple", "peach":"peach", "pie":"fruit", "potato":"peach"}`	
d) `{"cat":"cat", "corgi":"dog", "emu":"animal", "lab":"lair", "lair":"lair", "nyan":"cat"}`	

4. ADTs / Collections (write)

Write a function **`isSorted`** that accepts a reference to a stack of integers as a parameter and returns `true` if the elements in the stack occur in ascending (non-decreasing) order from top to bottom, else `false`. That is, the smallest element should be on top, growing larger toward the bottom. For example, passing the following stack should return `true`:

```
bottom {20, 20, 17, 11, 8, 8, 3, 2} top
```

The following stack is *not* sorted (the 15 is out of place), so passing it to your function should return a result of `false`:

```
bottom {18, 12, 15, 6, 1} top
```

An empty or one-element stack is considered to be sorted. When your function returns, the stack should be in the **same state** as when it was passed in. In other words, if your function modifies the stack, you must restore it before returning.

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

- You may use **one queue or one stack** (but not both) as auxiliary storage.
  You may not use other structures (arrays, lists, etc.), but you can have as many simple variables as you like.
- Your solution should run in O(*N*) time, where *N* is the number of elements of the stack.

---

5. Algorithm Analysis and 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 the variable N. In other words, write the code's 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 on the right side in the blanks provided.

Question	Answer

a)
`int sum = 0;`
`for (int i = 1; i <= N - 2; i++) {`
`  for (int j = 1; j <= i + 4; j++) {`
`  sum++;`
`  }`
`  sum++;`
`}`
 	
`for (int i = 1; i <= 100; i++) {`
`  sum++;`
`}`
`cout << sum << endl;`
	$O($_____$)$

b)
`int sum = 0;`
`for (int i = 1; i <= N; i++) {`
`  for (int j = 1; j <= N * N; j++) {`
`  sum++;`
`  }`
`  for (int j = 1; j <= 100; j++) {`
`  sum++;`
`  }`
`  for (int j = 1; j <= N; j++) {`
`  sum++;`
`  }`
`  sum++;`
`}`
`cout << sum << endl;`
	$O($_____$)$

c)
`int sum = 0;`
`for (int i = 1; i <= N; i++) {`
`  for (int j = 1; j <= 100; j++) {`
`  sum++;`
`  }`
`}`
 	
`for (int k = 1; k <= 10000; k++) {`
`  sum++;`
`}`
`cout << sum << endl;`
	$O($_____$)$

d)
`Set<int> set;`
`for (int i = 1; i <= N * 2; i++) {`
`  set.add(i);`
`}`
 	
`for (int k : set) {`
`  cout << k << endl;`
`}`
`cout << "done!" << endl;`
	$O($_____$)$

6. Recursion (read)

For each of the calls to the following recursive function below, indicate what output is produced:

```
void recursionMystery2(int n) {
    if (n <= 1) {
        cout << ": ";
    } else {
        cout << (n % 2) << " ";
        recursionMystery2(n / 2);
        cout << n << " ";
    }
}
```

Call	Output
a) `recursionMystery2(8);`	
b) `recursionMystery2(25);`	
c) `recursionMystery2(46);`	

7. Recursion (write)

Write a recursive function **`largestDigit`** that accepts an integer parameter and returns the largest digit value that appears in that integer. Your function should work for both positive and negative numbers. If a number contains only a single digit, that digit's value is by definition the largest. The following table shows several example calls:

Call	Returns
`largestDigit(14263203)`	`6`
`largestDigit(845)`	`8`
`largestDigit(52649)`	`9`
`largestDigit(3)`	`3`
`largestDigit(0)`	`0`
`largestDigit(-573026)`	`7`
`largestDigit(-2)`	`2`

Constraints: For full credit, obey the following restrictions in your solution. A solution that disobeys them gets partial credit.

- You may not use a `string`, array, `Vector`, or any data structure (stack, map, set, etc.).
- Your function must be recursive and not use any loops (`for`, `while`, etc.).
- Your solution should run in no worse than O(*N*) time, where *N* is the number of digits in the number.

8. Recursive Backtracking (write)

Write a recursive function **printSquares** that accepts an integer *n* as a parameter and uses backtracking to find all ways to express that integer *n* as a sum of squares of unique positive integers. For example, the call of `printSquares(200);` should produce the following output:

```
1^2 + 2^2 + 3^2 + 4^2 + 5^2 + 8^2 + 9^2
1^2 + 2^2 + 3^2 + 4^2 + 7^2 + 11^2
1^2 + 2^2 + 5^2 + 7^2 + 11^2
1^2 + 3^2 + 4^2 + 5^2 + 6^2 + 7^2 + 8^2
1^2 + 3^2 + 4^2 + 5^2 + 7^2 + 10^2
2^2 + 4^2 + 6^2 + 12^2
2^2 + 14^2
3^2 + 5^2 + 6^2 + 7^2 + 9^2
6^2 + 8^2 + 10^2
```

Some numbers (such as 128 or 0) cannot be represented as a sum of squares, in which case your function should produce no output. Keep in mind that the sum has to be formed with unique integers. Otherwise you could always find a solution by adding `1^2` together until you got to whatever total amount *n* you are working with.

If the value of *n* passed is negative, your function should throw an integer exception.

You may use a loop in your solution if you like, but the overall algorithm must use recursion and backtracking. You may define helper functions if you like.

To help you solve this problem, assume there already exists a function `display` that accepts any collection of integers (such as a vector, set, stack, queue, etc.) and prints the collection's elements in order in the format below. For example, if a vector `v` stores the elements `{1, 4, 8, 11}`, the call of `display(v);` would produce the following output:

```
1^2 + 4^2 + 8^2 + 11^2
```

9. Implementing a Collection Class (write)

Write a member function **mirror** to be added to the ArrayList class from lecture. Your function should double the size of the list of integers by appending the mirror image of the original sequence to the end of the list. The mirror image is the same sequence of values in reverse order. For example, suppose a variable called **list** stores the following values:

```
{1, 3, 2, 7}
```

If we make the call of **list.mirror();** then it should store the following values after the call:

```
{1, 3, 2, 7, 7, 2, 3, 1}
```

The list has been doubled in size by having the original sequence appearing in reverse order at the end of the list. Do not make assumptions about how many elements are in the list or about the list's capacity. If the list is empty, it should also be empty after the call. You may call other private member functions of the ArrayList if you like, but not public ones.

You should write the member function's body as it would appear in ArrayList.cpp. You do not need to write the function's header as it would appear in ArrayList.h. Write only your member function, not the rest of the class. Remember that you are adding this as a member function to the ArrayList class from lecture:

```
class ArrayList {
public:
    void add(int value);
    void clear();
    int get(int index) const;
    void insert(int index, int value);
    bool isEmpty() const;
    void remove(int index);
    void set(int index, int value);
    int size() const;
    string toString() const;
}
```

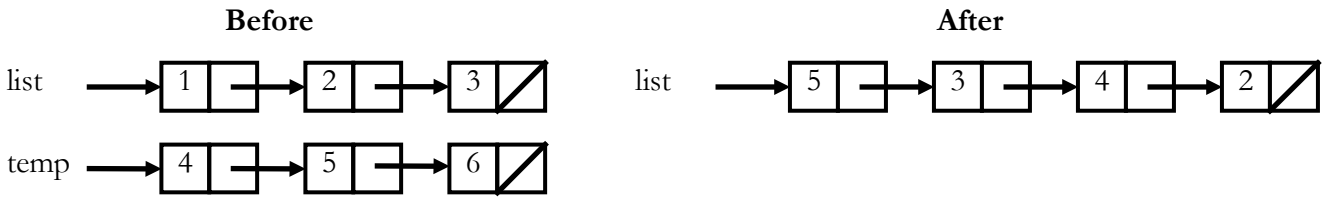
```
private:
    int* elements;
    int mysize;
    int capacity;
    void checkCapacity();
    void checkIndex(int i, int min, int max);
```

10. Pointers and Linked Nodes (write)

Write the code that will turn the "before" picture into the "after" picture by modifying links between the nodes shown. There may be more than one way to write the code, but you are *not* allowed to change any existing node's `data` field value. You also should *not* create new `ListNode` objects to add new values to a chain, but you may create **a single `ListNode*` pointer** variable to point to any existing node if you like.

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 *node* 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 solution code for the linked list section handout.



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

    ListNode(int data, ListNode* next) { ... }   // constructor
};
```