CS 106B Practice Midterm Exam #3

(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 code C++ uses parameters and produces four lines of output. What is the output?
For the purposes of this problem, assume that the variables in `main` are stored at the following memory addresses:

- `main`'s `w` variable is stored at address `0xaa00`
- `main`'s `x` variable is stored at address `0xbb00`
- `main`'s `y` variable is stored at address `0xcc00`
- `main`'s `z` variable is stored at address `0xdd00`

```
void parameterMystery3(int* a, int& b, int* c) {
    *a += *c;
    (*c)++;
    b--;
    cout << b << " " << *a << " " << *c << endl;
}

int main() {
    int w = 0;
    int x = 1;
    int y = 3;
    int z = 7;

    parameterMystery3(&y, x, &w);
    parameterMystery3(&x, y, &z);
    parameterMystery3(&w, w, &z);

    cout << w << " " << x << " " << y << " " << z << endl;
    return 0;
}
```

2. File I/O and Strings (write)

Write a function named **coinFlip** that accepts as its parameter a string holding a file name, opens that file and reads its contents as a sequence of whitespace-separated tokens. Assume that the input file data represents results of sets of coin flips. A coin flip is either the letter H or T, or the word Heads or Tails, in either upper or lower case, separated by at least one space. You should read the sequence of coin flips and output to the console the number of heads and the percentage of heads in that line, rounded to the nearest whole number. If this percentage is 50% or greater, you should print a "You win!" message; otherwise, print "You lose!". For example, consider the following input file:

```
H  T H  H    T
Tails taIlS   tAILs TailS heads   HEAds
   hEadS
```

For the input above, your function should produce the following output:

```
6 heads (50%)
You win!
```

The format of your output must exactly match that shown above. You may assume that the file contains at least 1 token of input, and that no tokens other than heads/tails or H/T will be in the lines.

If the input file does not exist or is not readable, your function should print no output.

---

3. ADTs / Collections (read)

Consider the following function:

```
void collectionMystery3(Queue<int>& q) {
    Stack<int> s;
    int size = q.size();
    for (int i = 0; i < size; i++) {
        int n = q.dequeue();
        if (n % 2 == 0) {
            s.push(n);
        } else {
            q.enqueue(n);
        }
    }
    cout << "q=" << q << endl;
    cout << "s=" << s << endl;
}
```

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

Queue	Output
a) {1, 2, 3, 4, 5, 6}	
b) {42, -3, 4, 15, 9, 71}	
c) {30, 20, 10, 60, 50, 40, 3, 0}	

4. ADTs / Collections (write)

Write a function **removeBadPairs** that accepts a reference to a `Vector` of integers and removes any adjacent pair of integers in the list if the left element of the pair is larger than the right element of the pair. Every pair's left element is an even-numbered index in the list, and every pair's right element is an odd index in the list. For example, suppose a variable called `vec` stores the following element values:

```
{3, 7, 9, 2, 5, 5, 8, 5, 6, 3, 4, 7, 3, 1}
```

We can think of this list as a sequence of pairs:

```
{3, 7, 9, 2, 5, 5, 8, 5, 6, 3, 4, 7, 3, 1}
```

The pairs 9-2, 8-5, 6-3, and 3-1 are "bad" because the left element is larger than the right one, so these pairs should be removed. So the call of `removeBadPairs(vec);` would change the vector to store:

```
{3, 7, 5, 5, 4, 7}
```

If the vector has an odd length, the last element is not part of a pair and is also considered "bad;" it should therefore be removed by your function.

If an empty vector is passed in, the vector should still be empty at the end of the call. Do not use any other arrays, vectors, or other data structures to help solve this problem, though you can create as many simple variables as you like.

---

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)
`HashSet<int> set1;`
`for (int i = 0; i < N; i++) {`
`  set1.add(i);`
`}`
`Set<int> set2;`
`for (int n : set1) {`
`  set2.add(n);`
`}`
`cout << "done!" << endl;`
	O(_____)

b)
`Vector<int> list;`
`for (int i = 0; i < N; i++) {`
`  list.insert(0, i*i);`
`}`
`Set<int> set;`
`for (int k : list) {`
`  set.add(k);`
`}`
`cout << "done!" << endl;`
	O(_____)

c)
`Vector<int> list1;`
`for (int i = 0; i < N; i += 2) {`
`  list1.add(i);`
`}`
`Vector<int> list2;`
`for (int i = 0; i < N; i++) {`
`  list2.insert(0, list1[0]);`
`  list1.remove(0);`
`}`
`cout << "done!" << endl;`
	O(_____)

d)
`int sum = 0;`
`for (int i = 0; i < N * 2; i++) {`
`  for (int j = 0; j < 100; j++) {`
`  for (int k = 0; k < j*j*j; k++) {`
`  sum++;`
`  }`
`  }`
`}`
`cout << sum << endl;`
	O(_____)

e)
`int sum = 0;`
`for (int i = 0; i < N * 2; i++) {`
`  for (int j = 0; j < i/2; j++) {`
`  for (int k = 0; k < N*N; k++) {`
`  sum++;`
`  }`
`  }`
`}`
`cout << sum << endl;`
	O(_____)

6. Recursion (read)

For each of the calls to the following recursive function below, indicate what value is returned:

```
int recursionMystery3(int n) {
    if (n < 0) {
        return -recursionMystery3(-n);
    } else if (n < 10) {
        return (n + 1) % 10;
    } else {
        return 100 * recursionMystery3(n / 10) + (n + 1) % 10;
    }
}
```

Call	Returns
a) `recursionMystery3(7)`	
b) `recursionMystery3(42)`	
c) `recursionMystery3(385)`	
d) `recursionMystery3(-790)`	
e) `recursionMystery3(89294)`	

7. Recursion (write)

Write a recursive function **matchCount** that accepts two references to vectors of integers as parameters and that returns the number of elements that match between them. Two elements match if they occur at the same index in both vectors and have equal values. For example, given the two vectors shown below, the call of `matchCount(v1, v2)` would compare as follows:

```
v1: {2, 5, 0, 3, 8, 9, 1, 1, 0, 7}
     |  |  |  |  |  |  |
v2: {2, 5, 3, 0, 8, 4, 1}
```

The function should return 4 in this case because 4 of these pairs match (2-2, 5-5, 8-8, and 1-1). If either vector is empty, by definition it has no matches with the other vector, so your function should return 0.

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

- Your function must be recursive and not use any loops (`for`, `while`, etc.).
- You may not use a `string`, array, or any data structure (stack, map, set, etc.) other than the vectors passed.
- When your code is done running, the two vectors should have the same contents as when the call began. Either do not modify the vectors, or if you do modify them, restore them to their original state afterward. Note again that you may not declare any additional data structures.

If it is helpful to you to solve this problem, you can declare additional helper functions that are utilized by `matchCount`, but your `matchCount` function itself must accept only the two `Vector`s as its parameters; do not change its header.

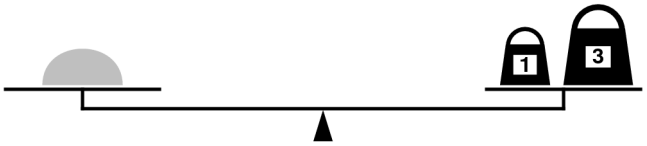
---

8. Recursive Backtracking (write)

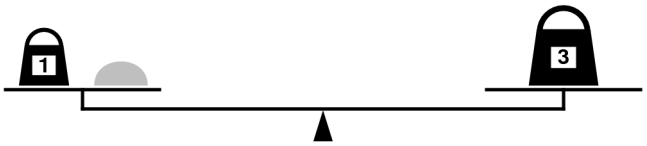
(thanks to Keith Schwarz for this problem)

Write a recursive function named **isMeasurable** that determines whether it is possible to weigh out exactly a given amount on two scales, given a target amount and a collection of weights to use. Your function accepts two parameters: a reference to a vector of integers representing weights that you can use, and an integer target weight amount.

For example, suppose that you have only two weights: a 1-ounce weight and a 3-ounce weight. With these you can easily measure out 4 ounces by placing both weights on one side of the scale, as shown:



You can also measure out 2 ounces by shifting the 1-ounce weight to the other side, as follows:



Suppose that the variable `weights` has been initialized as follows:

```
Vector<int> weights;
weights += 1, 3;
```

Given these values, the calls of `isMeasurable(weights, 4)` and `isMeasurable(weights, 2)` would both return `true`. On the other hand, the call of `isMeasurable(weights, 5)` would return `false` because it is impossible to use the 1- and 3-ounce weights to add up to 5 ounces. You don't need to use every weight in the vector to reach the given target; in the example above, calls of `isMeasurable(weights, 3)` and `isMeasurable(weights, 1)` would return `true` because you can make those amounts by using only a single weight in each case.

If the target weight passed is `0`, your function should always return `true`. You may assume that the value of the target weight passed will not be initially negative, and that the vector passed does not contain any negative weight values in it.

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

---

9. Implementing a Collection Class (write)

Write a member function **`longestSortedSequeunce`** to be added to the `ArrayList` class from lecture. Your function should return the length of the longest sorted sequence within the list of integers. For example, if a variable called `list` stores the following sequence of values:

```
{1, 3, 5, 2, 9, 7, -3, 0, 42, 308, 17}
```

Then the call of `list.longestSortedSequence()` would return `4` because it is the length of the longest sorted sequence within this list (the sequence -3, 0, 42, 308). If the list is empty, your method should return `0`. Notice that for a non-empty list the method will always return a value of at least 1 because any individual element is a sorted sequence.

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 unless necessary to add new values to the 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.

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
};
```