CS 106B Practice Midterm Exam #8

(based on CS 106B Spring 2015 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 stacks. Write the output exactly as it would appear on the console.

Note : A Stack prints in {bottom ... top} order.

```
void collectionMystery8(Stack<int>& stack) {
    Queue<int> queue;
    Set<int> set;
    while (!stack.isEmpty()) {
        if (stack.peek() % 2 == 0) {
            queue.enqueue(stack.pop());
        } else {
            set.add(stack.pop());
        }
    }
    for (int n : set) {
        stack.push(n);
    }
    while (!queue.isEmpty()) {
        stack.push(queue.dequeue());
    }
    cout << stack << endl;
}
```

---

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

b) {3, 2, 7, 3, 3, 4, 1, 1, 4}

c) {9, 7, 14, 7, 22, 7, 3, 14}

d) {8, 5, 1, 2, 1, 1, 2, 1, 4, 5}

2. Collections (write)

Write a function named **wordChain** that accepts two string parameters, the first representing an input filename and the second representing a starting word, and that produces a random "word chain" starting from the given word.

For this problem let's define a "word chain" as a sequence of words where the last two letters of the current word will be the first two letters of the next word. For example, here is a possible word chain that starts from the word "program":

program → amender → erected → edaciousness

The words you use in your word chain come from the given **input file**. You should assume that this file contains a sequence of words, one per line. Your function should open and read the contents of this input file and use those words when creating a word chain. If the file exists and is readable, you may assume that its contents consist entirely of words listed one-per-line in lowercase, and that each word in the dictionary is at least 2 letters long. For example, the file `dictionary.txt` might contain words in the format shown at right (abbreviated by ...).

```
aah
aahed
...
zygoid
zygote
```

You are producing a **random word chain**, so the idea is that you should randomly choose a next word whose first two letters are the same as the last two letters of the current word. In our sample chain above, any word starting with "am" would be a valid choice for the second word; and if the second word chosen is "amender", then any word starting with "er" would be a valid choice for the third word. And so on. A word chain might have a duplicate in it; this is okay.

A word chain **ends** when you reach a two-letter word suffix that is not the start of any word in the dictionary. For example, in the chain shown above, we produced "edaciousness". There are no words in the dictionary that begin with "ss", so the chain ends and the function stops.

Your function should print the word chain to the console, one word per line. Here are several example outputs from the call of `wordChain("dictionary.txt", "computer");` . The implication of the outputs below is that the given dictionary does not contain any words that begin with `"gs"`, or `"ss"`, or `"ns"`, which is why the chains end there.

```
computer
erecting
ngatis
isocyanates
esthete
terminism
smug
uglying
ngati
tidings
```

```
computer
ere
reservednesses
espouse
serovar
arpeggio
iodines
eschalot
ototoxicity
tyeing
nganas
assentive
vestibule
lecherousness
```

```
computer
erotize
zecchins
```

Notice that the same word suffix/prefix could occur more than once in the same chain, such as `"ng"` in `"erecting"` / `"ngatis"` and again later in `"uglying"` / `"ngati"`. If the start word passed in ends with a two-letter sequence that is not the start of any words in the input file, your function should simply print the start word and then exit.

If the file is missing/unreadable, or the start word's length is less than 2, your function should throw a string **exception**.

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.
  Similarly, don't store the entire file contents in a collection and loop over that entire collection multiple times.
- You should choose an **efficient** solution. Choose data structures intelligently and use them properly.
- 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.)

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

b)
`Vector<int> v;`
`for (int i = 0; i < N; i++) {`
`  v.insert(0, i);`
`}`
`while (!v.isEmpty()) {`
`  v.remove(0);`
`}`
`cout << "done!" << endl;`
	O(_____)

c)
`Queue<int> queue;`
`for (int i = 1; i <= N; i++) {`
`  queue.enqueue(i * i);`
`}`
`Map<int, int> map;`
`while (!queue.isEmpty()) {`
`  int k = queue.dequeue();`
`  map.put(k, N * N);`
`}`
`cout << "done!" << endl;`
	O(_____)

d)
`HashSet<int> set;`
`for (int i = 0; i < N; i++) {`
`  set.add(i);`
`}`
`Stack<int> stack;`
`for (int i = 0; i < N * N; i++) {`
`  stack.push(i);`
`}`
`for (int i = 0; i < N; i++) {`
`  set.remove(i);`
`  stack.pop();`
`}`
`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. *Hint:* To avoid re-tracing out long chains of calls, it may help you to look at your own notes on call results that you have already previously computed, and re-use those results as applicable.

```
void recursionMystery8(int n) {
    if (n <= 1) {
        cout << n;
    } else {
        cout << n << " = (";
        recursionMystery8(n / 2 + n % 2);
        cout << ", ";
        recursionMystery8(n / 2);
        cout << ")";
    }
}
```

Call	Output
a) `recursionMystery8(3);`	
b) `recursionMystery8(4);`	
c) `recursionMystery8(6);`	
d) `recursionMystery8(7);`	

5. Recursion (write)

Write a recursive function named **`collapseSequences`** that accepts a string `s` and char `c` as parameters and returns a new string that is the same as `s` but with any sequences of consecutive occurrences of `c` compressed into a single occurrence of `c`. For example, if we collapse sequences of character `'a'` in the string `"aabaaccaaaaada"`, you get `"abaccada"`.

Your function is case-sensitive; if the character `c` is, for example, a lowercase `'f'`, your function should not collapse sequences of uppercase `'F'` characters. In other words, you do not need to write code to handle case issues in this problem.

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

Call	Return Value
`collapseSequences("aabaaccaaaaaada", 'a')`	`"abaccada"`
`collapseSequences("mississssipppi", 's')`	`"misisipppi"`
`collapseSequences("babbbbebbbxbbbbbb", 'b')`	`"babebxb"`
`collapseSequences("palo alto", 'o');`	`"palo alto"`
`collapseSequences("tennessee", 'x')`	`"tennessee"`
`collapseSequences("", 'T')`	`""`

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 call** any of the following string functions: `find`, `rfind`, `indexOf`, `contains`, `replace`, `split`.
  (The point of this problem is to solve it recursively; do not use a library function to get around recursion.)
- 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, as well as `string`s.
- 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 **largestSum** that accepts a reference to a vector of integers *V* and an integer limit *N* as parameters and uses backtracking to find the largest sum that can be generated by adding elements of *V* that does not exceed *N*. For example, if you are given the vector `{7, 30, 8, 22, 6, 1, 14}` and the limit of `19`, the largest sum that can be generated that does not exceed is `16`, achieved by adding 7, 8, and 1. If the vector is empty, or if the limit is not a positive integer, or all of *V*'s values exceed the limit, return `0`. Assume that all values in the vector are non-negative.

Each index's element in the vector can be added to the sum only once, but the same number value might occur more than once in the vector, in which case each occurrence might be added to the sum. For example, if the vector is `{6, 2, 1}` you may use up to one `6` in the sum, but if the vector is `{6, 2, 6, 1}` you may use up to two sixes.

Here are several example calls to your function and their expected return values. The elements of the vector that combine to form that largest sum are underlined for clarity.

Vector *V*	Limit *N*	largestSum(*V*, *N*) returns
{**7**, 30, **8**, 22, 6, **1**, 14}	19	16
{**5**, 30, **15**, **13**, **8**}	42	41
{30, **15**, **20**}	40	35
{**6**, **2**, **6**, **9**, **1**}	30	24
{11, **5**, 3, **7**, **2**}	14	14
{10, 20, 30}	7	0
{10, **20**, 30}	20	20
{}	10	0

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**.
- You may not use any auxiliary **data structures**.
- Your code can contain **loops** if appropriate to solve the problem, but your overall algorithm must be recursive.
- The contents of vector V must be the same when your function returns. That is, your function may alter the contents of vector *V* as it executes, but if so, *V* should be restored to its original state before your function returns.
- You are allowed to define other **"helper"** functions if you like; they are subject to these same constraints.

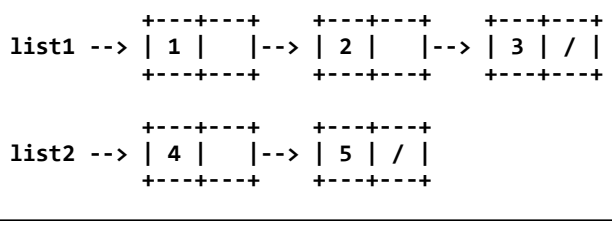
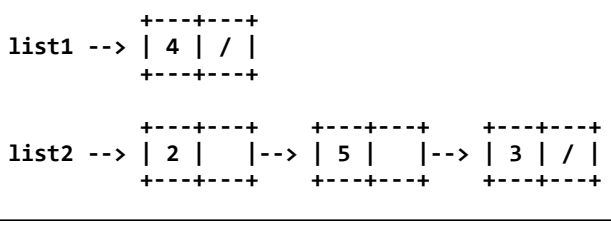
---

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 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 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
    ...
};
```