CS 106B Practice Midterm Exam #1

(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 <u>not</u> 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 `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`

```
int parameterMystery1(int a, int& b, int* c) {
    b++;
    a += *c;
    cout << b << " " << *c << " " << a << " " << c << endl;
    c = &a;
    return a - b;
}

int main() {
    int a = 4;
    int b = 8;
    int c = -3;
    int d;

    d = parameterMystery1(a, b, &c);
    parameterMystery1(c, d, &b);
    parameterMystery1(b, a, &d);
    cout << a << " " << b << " " << c << " " << d << endl;

    return 0;
}
```

2. File I/O and Strings (write)

Write a function named **`wordStats`** that accepts as its parameter a string holding a file name, opens that file and reads its contents as a sequence of words, and produces a particular group of statistics about the input. You should report the total number of words (as an integer), the average word length (as an un-rounded real number), and the number of unique letters used from A-Z, case-insensitively. For example, suppose the file `tobe.txt` contains the following text:

```
To be  or not  TO BE,    THAT IS   the question.
```

For the purposes of this problem, we will use whitespace to separate words. That means that some words include punctuation, as in "be,". For the input above, your function should produce exactly the following output. The number of "unique letters" is 12 because the file contains 12 distinct letters of the alphabet from A-Z: a, b, e, h, i, n, o, q, r, s, t, and u.

```
Total words    = 10
Average length = 3.2
Unique letters = 12
```

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

```
Error, bad input file.
```

For full credit, your solution should read the file only once, not make multiple passes over the file data.

---

3. ADTs / Collections (read)

Consider the following function:

```
void collectionMystery1(Vector<int>& list) {
    for (int i = 0; i < list.size(); i++) {
        int n = list[i];
        list.remove(i);
        if (n % 2 == 0) {
            list.add(i);
        }
    }
    cout << list << endl;
}
```

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

Vector	Output
a) `{5, 2, 5, 2}`	
b) `{3, 5, 8, 9, 2}`	
c) `{0, 1, 4, 3, 1, 3}`	

4. ADTs / Collections (write)

Write a function **byAge** that accepts three parameters: 1) a reference to a `Map` where each key is a person's name (a string) and the associated value is that person's age (an integer); 2) an integer for a minimum age; and 3) an integer for a max age. Your function should return a new map with information about people with ages between the min and max, inclusive.

In your result map, each key is an integer age, and the value for that key is a string with the names of all people at that age, separated by `"and"` if there is more than one person of that age. The order of names for a given age should be in alphabetical order, such as `"Bob and Carl"` rather than `"Carl and Bob"`. (This is the order in which they naturally occur in the parameter map.) Include only ages between the min and max inclusive, where there is at least one person of that age in the parameter map. If the map passed is empty, or if there are no people in the map between the min/max ages, return an empty map.

For example, if a `Map` named `ages` stores the following key:value pairs:

```
{"Allison":18, "Benson":48, "David":20, "Erik":20, "Galen":15, "Grace":25,
 "Helene":40, "Janette":18, "Jessica":35, "Marty":35, "Paul":28, "Sara":15,
 "Stuart":98, "Tyler":6, "Zack":20}
```

The call of `byAge(ages, 16, 25)` should return the following map:

```
{18:"Allison and Janette", 20:"David and Erik and Zack", 25:"Grace"}
```

For the same map, the call of `byAge(ages, 20, 40)` should return the following map:

```
{20:"David and Erik and Zack", 25:"Grace", 28:"Paul", 35:"Jessica and Marty", 40:"Helene"}
```

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

- You will need to construct a map to store your results, but you may not use any other structures (arrays, lists, etc.) as auxiliary storage. (You can have as many simple variables as you like.)
- You should not modify the contents of the map passed to your function.
  Declare your function in such a way that any caller can be sure that this will not happen.
- Your solution should run in no worse than O(N log N) time, where N is the number of pairs in the map.

---

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++) {`
`  sum++;`
`}`
`for (int j = 1; j <= N * 2; j++) {`
`  sum += 5;`
`}`
`cout << sum << endl;`
	O(_____)

b)
`int sum = 0;`
`for (int i = 1; i <= N - 5; i++) {`
`  for (int j = 1; j <= N - 5; j += 2) {`
`  sum++;`
`  }`
`}`
`cout << sum << endl;`
	O(_____)

c)
`int sum = N;`
`for (int i = 0; i < 1000000; i++) {`
`  for (int j = 1; j <= i; j++) {`
`  sum += N;`
`  }`
`  for (int j = 1; j <= i; j++) {`
`  sum += N;`
`  }`
`  for (int j = 1; j <= i; j++) {`
`  sum += N;`
`  }`
`}`
`cout << sum << endl;`
	O(_____)

d)
`Vector<int> list;`
`for (int i = 1; i <= N * N; i++) {`
`  for (int j = 1; j <= N; j++) {`
`  list.add(i + j);`
`  }`
`}`
 	
`for (int i = 1; i <= 2 * N; i++) {`
`  list.remove(list.size() - 1);`
`}`
 	
`cout << "done!" << endl;`
	O(_____)

e)
`HashSet<int> set1;`
`for (int i = 1; i <= N; i++) {`
`  set1.add(i);`
`}`
 	
`Set<int> set2;`
`for (int i = 1; i <= N; i++) {`
`  set1.remove(i);`
`  set2.add(i + N);`
`}`
`cout << "done!" << endl;`
	O(_____)

6. Recursion (read)

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

```
void recursionMystery1(int x, int y) {
    if (y <= 0) {
        cout << "0 ";
    } else if (x > y) {
        cout << x << " ";
        recursionMystery1(x - y, y);
    } else {
        recursionMystery1(x, y - x);
        cout << y << " ";
    }
}
```

Call	Output
a) `recursionMystery1(6, 3);`	
b) `recursionMystery1(2, 3);`	
c) `recursionMystery1(5, 8);`	
d) `recursionMystery1(21, 12);`	
e) `recursionMystery1(3, 10);`	

7. Recursion (write)

Write a recursive function **moveToEnd** that accepts a string *s* and a character *c* as parameters, and returns a new string similar to *s* but with all instances of *c* moved to the end of the string. The relative order of the other characters should be unchanged from their order in the original string *s*. If the character is a letter of the alphabet, all occurrences of that letter in either upper or lowercase should be moved to the end and converted to uppercase. If *s* does not contain *c*, it should be returned unmodified.

The following table shows calls to your function and their return values. Occurrences of *c* are underlined for clarity.

Call	Returns
moveToEnd("hello", 'l')	"heoLL"
moveToEnd("hello", 'e')	"hlloE"
moveToEnd("hello THERE", 'e')	"hllo THREEE"
moveToEnd("hello there", 'q')	"hello there"
moveToEnd("banAnA ramA", 'A')	"bnn rmAAAAA"
moveToEnd("x", 'x')	"X"
moveToEnd("", 'x')	""

You may not construct any data structures (no array, vector, stack, etc.), and you may not use any loops to solve this problem; you must use recursion. Also, you may not use any global variables in your solution.

---

8. Recursive Backtracking (write)

Write a function called **crack** that uses **recursive backtracking** to search for the correct password to break into a secure system. A common way of cracking users' passwords is to write a program that tries all possible passwords until one of them works. You will write a `crack` function that accepts an integer parameter representing a maximum password length. Your function will try all passwords up to the given length inclusive, searching for the right one. If your function finds the right password, it returns that string. If not, it returns an empty string. To simplify the problem, let's assume that passwords consist entirely of lowercase letters from `'a'` through `'z'`.

Suppose that the following function has already been defined, and is available for you to call as much as you like:

```
bool login(string password)
```

If you pass a string to the above function, it will return `true` if that password string logs you in successfully; in other words, if the string you passed it is the correct password. You can call the `login` function as many times as you want to try to find the right password. Your job is to exhaustively try *all* possible passwords until you find the correct one.

For example, let's suppose that the correct password is `"viva"`. The call of `crack(4)` would try generating all non-empty strings of letters up to length 4 and testing them as passwords. You might start with single-letter strings like `"a"` through `"z"`, then 2-letter strings like `"aa"`, `"ab"`, ..., `"zz"`, then `"aaa"`, `"aab"`, ..., `"zzz"`, then `"aaaa"`, `"aaab"`, ... and so on. Eventually you would try calling `login("viva")`, which would return `true`, so your algorithm should notice this and return `"viva"`. You can generate the strings in any order you like, as long as you generate them all properly. If we had called `crack(3)`, it would never try a 4-letter string like `"viva"`, so it would try all possible 1-letter through 3-letter passwords, none of which would succeed, causing it to eventually give up and return `""`.

If the maximum length passed is 0, return the empty string, `""`. If the max length is negative, throw an integer **exception**.

You may define **helper functions**, and you may use auxiliary data structures if you like, but the amount of memory you use should not grow exponentially with respect to the maximum length passed. In other words, don't store every single word you generate into a gigantic data structure, because this would use way too much memory.

Hint: You can loop over a range of characters much like looping over integers, using a standard `for` loop.
You can use **loops** in your code, as long as your overall algorithm is recursive and uses backtracking.

---

9. Implementing a Collection Class (write)

Write a member function **maxCount** to be added to the ArrayList class from lecture. Your function should examine the elements of the list and return the number of occurrences of the most frequently occurring value in the list of integers. For this problem, you should assume that the elements in the list are in sorted order. Because the list will be sorted, all duplicates will be grouped together, which will make it easier to count duplicates. For example, suppose that an ArrayList called list stores the following sequence of values:

```
{1, 3, 4, 7, 7, 7, 7, 9, 9, 11, 13, 14, 14, 14, 16, 16, 18, 19, 19, 19}
```

This list has some values that occur just once (1, 3, 4, 11, 13, 18), some values that occur twice (9, 16), some values that occur three times (14, 19) and a single value that occurs four times (7). Therefore, the client's call of `list.maxCount()` should return **4** to indicate that the most frequently occurring value occurs 4 times. It is possible that there will be a tie for the most frequently occurring value, but that doesn't affect the outcome because you are just returning the count, not the value. If there are no duplicates in the list, then every value will occur exactly once and the maximum would be 1. If the list is empty, you should return 0.

You may call other member functions of ArrayList if you like, but your function should not modify the state of the list. Its header should be declared in such a way as to indicate to the client that this function does not modify the list.

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;
}
```

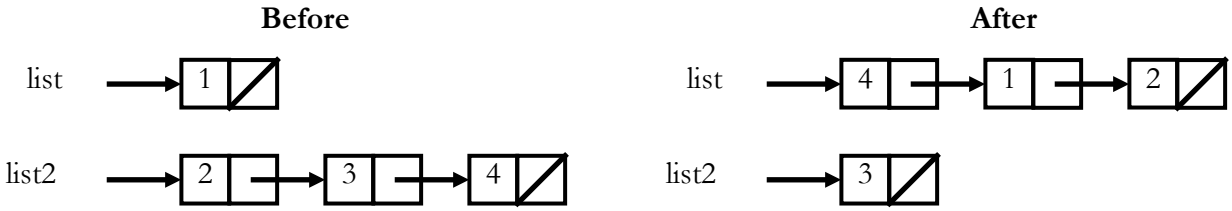
```
private:
    int* elements;
    int mysize;
    int capacity;
    void checkCapacity();
    void checkIndex(int i, int min, int max);
```

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 and/or creating new nodes as needed. 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.



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
};
```