CS106X

Instructor: Cynthia Lee

Autumn 2015

October 27, 2015

MIDTERM EXAM

NAME (LAST, FIRST): ____

SUNET ID: ____@stanford.edu

Problem	1	2	3	4	5	TOTAL
Topic	ADTs	Pointers, Memory	Classes	Recursion	Big-O	
Score						
Possible	34	18	30	26	12	120

Instructions:

- The time for this exam is **2 hours**. There are 120 points, or 1 point per minute. This gives you some general idea of how to pace yourself on each problem.
- Use of anything other than a pencil, eraser, pen, one 8.5x11 page (two sides) of notes, and the official textbook is prohibited. In particular, **no** computers or digital devices of any kind are permitted. Blank scratch paper may be provided by proctors and does not need to be turned in.
- PLEASE rip off the pages of library reference in the back of the exam and do not turn them in.
- Please do NOT staple or otherwise insert new pages into the exam. Changing the number of pages in the exam confuses our automatic scanning system. Thanks.
- SCPD and OAE: Please call or text 760-845-7489 if you have a question.

Please sign *before* you begin:

I agree to abide by the spirit and letter of the Honor Code, and to follow the instructions above.

____ ______________ ______________________

(Signature) (Date) (Start time - HH:MM zone)

1. **ADTs (34pts).** A classic of cryptographic methods is the substitution cipher. In a substitution cipher, each instance of a given letter is replaced throughout the text by some other letter, and this is done for all letters. So, for example, if we use the substitutions `d->Z`, `f->E`, `g->M`, `n->P`, `o->G`, `s->C`,[1] (and letters not named going to various other letters), you get this:

   [plaintext] `goodness` -> *[ciphertext]* `MGGZPFCC`

   Note that each letter must map to a unique other letter (so having both `c->Z` and `d->Z` in the same cipher is not allowed). One thing to note about substitution ciphers is that we can look at just the encoded text of a word, without any idea of what the substitutions might have been, and still rule out many words that could not have been the original plain text for it. For example, we know that "`MGGZPFCC`" could not have been "`bye`" because they are not the same length. We also know that "`MGGZPFCC`" could not have been "`puzzling`," because even though they are the same length, we don't see the repetition of some substituted letter for "`zz`" in the right place in the word. This **examination of patterns** of repeated and non-repeated letters is exactly what this problem will undertake.

   (a) (17pts) Write a function that, given a Lexicon (already initialized to a large set of English words, you may assume all in lowercase), produces a mapping of repetition patterns to a list of words matching that pattern. **The patterns will be produced by taking a word and replacing the first letter with `'A'`, the second unique letter with `'B'`, the third unique letter with `'C'`, and so on.** Here is a small example of what the map contents should be:

Pattern strings (map keys)	**Vectors of plaintext words (map values)**
"ABC"	{ "dog", "cot", "bar", "gum" }
"ABA"	{ "mum", "bob" }
"ABB"	{ "zoo", "see", "ebb" }
"ABBC"	{ "good", "beet" }
"ABACDEFG"	{ "elephant" }

- The required function signature is:
  `Map<string, Vector<string>> getPatterns(const Lexicon &english)`
- Suggestion: write a helper function that just does the pattern creation, because you'll want to use it in both Part(a) and Part(b). That way you only have to write that part once, and just call it in both.
- Reminder: You can "add one" to a char to get the next letter in the alphabet (`char c = 'A'; c++; // now c is 'B'`)
- Write your solution on the next page.
- Exam code isn't scrutinized for style the way homework submissions are, but for an ADTs problem you should pay attention to making good, clean, efficient use of the ADTs.
- In grading Part (b) of this problem, we will assume your Part (a) works as specified.

[1] To make it very clear when we are talking about plaintext (the original message) and when we are talking about the ciphertext or pattern strings, this problem will use all lowercase for plaintext and ALL CAPS for ciphertext and pattern strings.

```
Map<string, Vector<string>> getPatterns(const Lexicon &english) {









}
```

(b) (17pts) Once we have our Map from Part (a) created, we want to use it to give us clues for decoding ciphertexts by narrowing down the list of possible plaintexts for a given ciphertext word. Write a function that, **given** a ciphertext string, the Map from Part (a), and a Map giving any characters whose substitutions are already known (i.e., you already did some sleuthing and you are certain you know the correct cipher mapping for some characters), **returns** a list of the possible plaintext words that could correspond to the ciphertext. These examples all assume the Map is the one from the example in Part (a):

Example 1:

Function input: Ciphertext word: `"ZGZ"`, Known substitutions: g->L, q->R, k->F
Step 1: the pattern string for that ciphertext is `"ABA"`
Step 2: get this list out of the Map: `{ "mum", "bob" }`
Step 3: filter the list to remove any that are incompatible with the provided known substitutions (in this case both are compatible, so we return both).
Function returns: `{ "mum", "bob" }`

Example 2:

Function input: Ciphertext word: `"ZGZ"`, Known substitutions: b->Q, z->U
Step 1: the pattern string for that ciphertext is `"ABA"`
Step 2: get this list out of the Map: `{ "mum", "bob" }`
Step 3: filter the list to remove any that are incompatible with the provided known substitutions (in this case, `"bob"` is incompatible because we know b->Q).
Function returns: `{ "mum" }`

Example 3:

Function input: Ciphertext word: `"ZGZ"`, Known substitutions: o->G, c->D
Step 1: the pattern string for that ciphertext is `"ABA"`
Step 2: get this list out of the Map: `{ "mum", "bob" }`
Step 3: filter the list to remove any that are incompatible with the provided known substitutions (in this case, `"mum"` is incompatible because we know o->G).
Function returns: `{ "bob" }`

- The required function signature is:
```
Vector<string> getCandidatePlaintext(string ciphertextWord,
        const Map<string, Vector<string>> &patternToWords,
        const Map<char, char> &knownSubstitutions)
```
- The `patternToWords` Map is the return value from Part (a).
- The `knownSubstitutions` Map is structured so that the keys are the plaintext characters, and the values are the ciphtertext characters (e.g., `'o'->'G'`).
- Write your solution on the next page.
- Exam code isn't scrutinized for style the way homework submissions are, but for an ADTs problem you should pay attention to making good, clean, efficient use of the ADTs.
- In grading Part (b) of this problem, we will assume your Part (a) works as specified.

```
Vector<string> getCandidatePlaintext(string ciphertextWord,
                 const Map<string, Vector<string>> &patternToWords,
                 const Map<char, char> &knownSubstitutions){
```

```
}
```

2. **Pointers and Memory (18pts).** Draw the state of memory at the end of the execution of this code. Be careful in showing where your pointers originate and terminate (outer box vs. inner box). Leave uninitialized or unspecified areas blank, and clearly mark NULL (write NULL or draw a slash through the box). Draw the components in the appropriate stack and heap areas marked for you. Mark memory that has been deleted by enclosing it in a circle with a slash through it, like this: 🚫 but leave it where it is, and do not change any pointer or other values unless they are actually changed.

```
Doge soTest;
soTest.much = 3;
soTest.wow = new Doge[2];
soTest.wow[0].wow = new Doge;
soTest.wow[0].wow->much = 5;
// (a) Draw the state of memory now
```

```
struct Doge {
   int much;
   Doge *wow;
};
```

(a) (5pts) DRAWING:

Stack:

Heap:

(b) (4pts) In the space below, write the code that would be necessary to delete all the memory that needs deleting at this point (so there could not be any memory leaks later). Do not write any code that isn't strictly necessary for accomplishing this. Then, <u>update the above drawing</u> (in Part (a)) as if your lines of code have now been executed (i.e., mark any that memory was deleted by your code).

```
Doge *soMidterm = new Doge;
soMidterm->much = 7;
soMidterm->wow = new Doge[2];
Doge **muchPoint = &(soMidterm->wow);
// (c) Draw the state of memory now
```

```
struct Doge {
   int much;
   Doge *wow;
};
```

(c) (5pts) DRAWING:

Stack:

Heap:

(d) (4pts) In the above drawing (Part (c)), label any memory that is already orphaned "Orphaned by Part (c)." Then, in the space below, write one line of code that would result in some of the memory in the above drawing to be orphaned. DO **NOT** EDIT THE DRAWING TO SHOW THE RESULT OF EXECUTING YOUR CODE, except that you should circle all memory that your line of could *would* orphan, *if* we executed it, and label it "Would be orphaned by Part (d)."

3. **Classes (30pts).** You've decided to "disrupt" the Trick-or-Treating market by writing an app that keeps track of kids' candy while they go trick or treating. As part of the app, you need to write a CandyBag class to help you remember what candy you have picked up, how much you can carry, and what candy you have remaining.

   You are responsible for implementing the public interface given in the header file below, which includes the constructor, destructor, and public functions of the class. Each function has a comment that explains what it is responsible for doing as well as any special cases you need to handle. You should design your use of ADTs to provide an efficient implementation. Read the function comments carefully **before** beginning to code up any part of the class. We highly recommend that you plan out how you will implement these functions and how they will interact with your private member variables of the class before you begin writing any code on this problem.

   You should write the implementation of the public functions as if you were writing the .cpp file of the class, so make sure that you use the appropriate syntax. You should put any private class fields that you would like to add into the private section of the header file below.

   Please be sure to double-check that you implement all 7 functions, as well as the relevant private data in the candybag.h file.

```
//beginning of candybag.h file

//assume necessary #includes are done

class CandyBag {
public:
    struct CandyBar {  //assume all Stanford Library containers can hold this
        string type;
        double weight;
        CandyBar(string t="", double w=0.0) { type = t; weight = w; }
    };

    /* Create a CandyBag with the given weightLimit, which is the maximum
     * weight (bag weight plus candy weight) the candy bag can support.
     * Initially the bag has no candy. The variable bagWeight represents the
     * weight of the empty bag.
     */
    CandyBag(double weightLimit, double bagWeight);

    /* Delete any memory that this class is responsible for. */
    ~CandyBag();

    // candybag.h file continues on the next page
// candybag.h file, continued


    /* Get a list of all candy bar types we have stored in our CandyBag.
     * There should be no duplicates in this list. If the CandyBag is empty,
     * return an empty Vector.
     */
    Vector<string> getCandyBarTypes();

    /* Add a candy bar of the given type and weight to the CandyBag (note
     * that candy bars of the same type may come in different sizes and hence
     * have different weights). If there is no room for this candy bar (we
     * cannot carry this much more weight), then throw an error politely
     * declining the candy bar.
     */
    void addCandyBar(const string& type, double weight);

    /* Chooses a candy bar (any one is fine) from the CandyBag, removes it
     * from the CandyBag, and returns that candy bar. If there is no candy
     * left in the CandyBag, then throw an error.
     */
    CandyBar eatCandyBar();

    /* Returns the number of different weight candy bars of the given type
     * the CandyBag currently holds. Should run in constant time. If no
     * candies of this type are present in the CandyBag, return 0.
     */
    int getNumOfDistinctCandyTypes(const string& type);

    /* Returns the total weight of the candy of the given candy types
     * currently held in our candy bag. If no candies of these types are
     * present in the CandyBag, return 0.0.
     */
    double getWeightOfCandyTypes(const Set<string>& types);


// candybag.h file continues on the next page
// candybag.h file, continued

private:
    // TODO: Add any private fields or methods that you need for your
    //       implementation. You may not need this much space.









};
// End of candybag.h file


// Beginning of candybag.cpp file

#include "candybag.h"

// TODO: Add implementations of your methods here
```

// candybag.cpp file continues next page

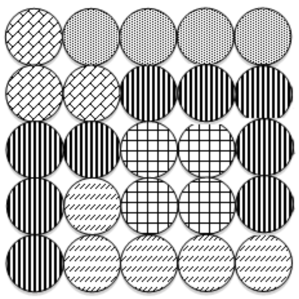
```
// candybag.cpp file, continued
```

```
// End of candybag.cpp file
```

4. **Recursion (26pts).** Recall from *Fauxtoshop* that images can be stored as a `Grid<int>`, where the individual pixels (row/col entries) of the Grid are integers representing RGB values (encoded colors). Your task is to implement a "magic wand selection tool" function. In some image editors, this feature is represented by a magic wand icon: 🪄. The magic wand selection tool selects all the pixels of the same (or similar) color within a contiguous region. The contiguous region can have any shape and any size; its boundaries are determined solely based on when a different color (or an edge of the Grid) is encountered. The user indicates one "start" pixel to select. Any adjacent (up/down/left/right) pixels that are the same (or similar) color are also selected, as well as any same-color (or similar to the *original start pixel* color) pixels adjacent to them, and so on.

   The required function signature is:
   ```
   Set<Point> magicWand(const Grid<int> &image, Point pixel, int threshold)
   ```

   Example:

   - This example uses the image at the right, where each pixel is one small circle, and different colors are represented by different patterns.
   - Given the input `Point pixel` with coordinates row=1, col=4, your function should return a Set containing the Point coordinates `{(1,2), (1,3), (1,4), (2,4), (3,4)}`. There are other areas of the image with the same "color," but they are not contiguous with the indicated pixel. We are assuming that other colors in the image are more than `threshold` difference, and thus not "very similar."

   Notes:
   - You may use the function signature given above as a wrapper for an actual recursive function that takes more parameters.
   - The Point struct is as follows[2]:
     ```
     struct Point {
         int row;
         int col;
         Point(int r=0, int c=0) { row = r; col = c; }
     };
     ```
   - To test if two colors are "very similar," you should call a function `int pixelDiff(int color1, int color2)`. Assume this function is already defined for you, and just call it. The order of arguments does not matter. You will consider two colors "very similar" if their distance, according to `pixelDiff`'s return value, is strictly less than `threshold`.
   - The shape defined by a given point includes the given point and all values that are immediately adjacent **(up, down, left, right, *NOT diagonal*)** to some other pixel in the shape, and have the same or similar color *to the **original** starting pixel*.

[2] You may assume that operator == works on Points in an intuitive way, and that Points can be stored in any Stanford Library container.

```
Set<Point> magicWand(const Grid<int> &image, Point pixel, int threshold) {

}
```

5. **Big-O (12pts).** Give a tight bound of the nearest runtime complexity class (worst-case) for each of the following code fragments in Big-O notation, in terms of variable N. As a reminder, when doing Big-O analysis, we write a simple expression, such as O(N^2) or O(log N), *not* an exact calculation. Write your answer in the blanks on the right side.

Question (3pts each)	Answer

`void myfunction(const Vector<int> &vec){`
`    int N = vec.size(); //assume N > 5`
`    for(int i = 0; i < vec.size(); i += (vec.size() / 5)) {`
`        for (int j = 0; j < vec.size(); j++) {`
`            for (int k = 0; k < vec.size() / 2; k++) {`
`                cout << vec[i] << endl;`
`            }`
`            for (int k = 0; k < vec.size() / 4; k++) {`
`                cout << vec[i] << endl;`
`            }`
`        }`
`    }`
`}`
	O( )

`int foofind(const Vector<int> &vec, int key) {`
`    int N = vec.size(); //assume N > 3`
`    int index = -1;`
`    for(int i = 0; i < vec.size(); i += 3) {`
`        int j = 0;`
`        while (vec[j] < key && j < N) {`
`            j++;`
`        }`
`        index = j - 1;`
`    }`
`    return index;`
`}`
	O( )

`void myfunction(const Vector<int> &vec) {`
`    int N = vec.size(); //assume N > 2`
`    for (int i = 0; i < N / 2; i++) {`
`        for (int j = i; j < N / 2; j++) { // note j = i`
`            cout << vec[j] << endl;`
`        }`
`    }`
`}`
	O( )

`void divider(int N) { // assume N > 2`
`    while (N > 2) {`
`        cout << (N /= 2) << endl;`
`    }`
`}`
	O( )

Summary of Relevant Data Types
We tried to include the most relevant member functions and operators for the exam, but not all are listed. You are free to use ones not listed here that you know exist. ***You do <u>not</u> need to do #include for these.***

```
class string {
 bool empty() const;
 int size() const;
 int find(char ch) const;
 int find(char ch, int start) const;
 string substr(int start) const;
 string substr(int start, int length) const;
 char& operator[](int index);
 const char& operator[](int index) const;
};

class Vector {
 bool isEmpty() const;
 int size() const;
 void add(const Type& elem); // operator += used similarly
 void insert(int pos, const Type& elem);
 void remove(int pos);
 Type& operator[](int pos);
};

class Grid {
 int numRows() const;
 int numCols() const;
 bool inBounds(int row, int col) const;
 Type get(int row, int col) const; // cascade of operator [] also works
 void set(int row, int col, const Type& elem);
};

class Stack {
 bool isEmpty() const;
 void push(const Type& elem);
 Type pop();
};

class Queue {
 bool isEmpty() const;
 void enqueue(const Type& elem);
 Type dequeue();
};
```

```
class Map {
 Vector<KeyType> keys() const;
 bool isEmpty() const;
 int size() const;
 void put(const Key& key, const Value& value);
 bool containsKey(const Key& key) const;
 Value get(const Key& key) const;
 Value& operator[](const Key& key);
};
```
Example range-based for: `for (Key key : mymap){…}`

```
class Set {
 bool isEmpty() const;
 int size() const;
 void add(const Type& elem); // operators + and += also add elements
 // operator + between sets is a set union operation, e.g., set1 + set2
 bool contains(const Type& elem) const;
};
```
Example range-based for: `for (Type elem : myset){…}`

```
class Lexicon {
  int size() const;
  bool isEmpty() const;
  void clear();
  void add(std::string word);
  bool contains(std::string word) const;
  bool containsPrefix(std::string prefix) const;
};
```
Example range-based for: `for (string str : english){…}`