Practice Midterm 2

CS106B

_____

(Print name legibly)

_____

(SUID **number**)

Exam Instructions

There are **5** questions worth a total of **100** points. Write all answers directly on the exam paper. This printed exam is closed-book and closed-device; you may refer only to our provided reference sheet. You are required to write your SUID number in the blank at the top of each odd-numbered page.

C++ Coding Guidelines

Unless otherwise restricted in the instructors for a specific problem, you are free to use any of the CS106B libraries and classes. You don't need *#include* statements in your solutions; just assume the required header files (*vector.h*, *strlib.h*, etc.) are visible. You do not need to declare prototypes. You are free to create helper functions unless the problem states otherwise. Comments are not required, but when your code is incorrect, comments could clarify your intentions and help the graders award partial credit.

The Stanford University Honor Code (2023 Revision)

The Honor Code is an undertaking of the Stanford academic community, individually and collectively. Its purpose is to uphold a culture of academic honesty.

Students will support this culture of academic honesty by neither giving nor accepting unpermitted academic aid in any work that serves as a component of grading or evaluation, including assignments, examinations, and research.

Instructors will support this culture of academic honesty by providing clear guidance, both in their course syllabi and in response to student questions, on what constitutes permitted and unpermitted aid. Instructors will also not take unusual or unreasonable precautions to prevent academic dishonesty.

Students and instructors will also cultivate an environment conducive to academic integrity. While instructors alone set academic requirements, the Honor Code is a community undertaking that requires students and instructors to work together to ensure conditions that support academic integrity.

In signing below, I acknowledge, accept, and agree to abide by the Honor Code.

_____

(signature)

This cover sheet has been revised to reflect the format you should expect to see on this quarter's exam.

1. C++ and ADTs (30 pts)

Moira's Teashop of Wonderment and Whimsy serves phenomenal teas with unusual names. They are working on revising their *extensive* menu of teas and would like to quickly look up all the teas on their menu whose descriptions contain certain phrases. To help, you will write a function with the following signature (as well as two helpers described below):

```
Set<string> matches(Map<int, Vector<Vector<string>>>& teaDescriptions,
                    Map<string, int>& teaNamesToIDs, Vector<string>& searchPhrase)
```

The parameters are as follows:

- `Map<int, Vector<Vector<string>>>& teaDescriptions`
  A map where each key is an integer that Moira's Teashop uses as a unique ID for one of their teas, and each value is a vector of string vectors, with one string vector per sentence in the tea's menu description. Each vector of strings corresponds to a single sentence and contains the words of that sentence, in order, converted to lowercase, with all punctuation removed. For example, suppose the description for tea 42 is: "A creamy chocolate rooibos tea. Organic, herbal, caffeine free. Available hot or iced." If this were the only item on the menu, the map would look as follows, with 42 mapping to the corresponding vector of string vectors:

  ```
  {
    42 : {
          { "a", "creamy", "chocolate", "rooibos", "tea" },
          { "organic", "herbal", "caffeine", "free" },
          { "available", "hot", "or", "iced" }
    }
  }
  ```

- `Map<string, int>& teaNamesToIDs`

  A map where each key is the (unique) name of one of the teas on the menu, and each value is the (unique) integer ID used to identify that tea in the map of tea descriptions above. For example:

  ```
  {
    "Tea of Jollity" : 1,
    "Tea of Jubilation" : 1234,
    "Tea of Wonderment" : 42,
    "Tea of Lucid Dreams" : 333
  }
  ```

- `Vector<string>& searchPhrase`

  A vector corresponding to a phrase we want to search for in the menu item descriptions. As with the menu item descriptions themselves, this vector contains the words of the phrase we're searching for, in order, converted to lowercase, with all punctuation removed. So, if someone is searching for the phrase "Caffeine free!", we can assume that phrase has been pre-processed elsewhere and that the vector passed to this function will be:

  ```
  { "caffeine", "free" }
  ```

The *matches()* function should return a set of the names of **all** teas (as strings) whose menu descriptions match the given phrase. So, if we call *matches(teaDescriptions, teaNames, {"chocolate", "rooibos"})*, based on the values above, our resulting set should contain the string "Tea of Wonderment" (as well as the names of any other teas whose descriptions contain the phrase "chocolate rooibos").

Here are some restrictions:

- Your functions must be **iterative** (not recursive) to earn credit.
- You must use the function signatures given below.
- Other than the **three** functions whose signatures are given below, you cannot write any **helper functions**.
- Please be careful not to go out of bounds in any vectors.
- You must not modify any of the parameters passed to these functions by reference.
- You may assume all tea IDs are unique, as are the names of the teas.

To make the code more readable, you must also write – and *call* -- the helper functions described below. Part of this problem is to figure out how to call these helper function appropriately from within your *matches()* function.

a) This function returns *true* if *searchPhrase* is found within the given *sentence* vector, *false* otherwise. All strings from *searchPhrase* must be found in *sentence*, consecutively and in the same order in which they appear in *searchPhrase*.

```
bool containsPhrase(Vector<string>& sentence, Vector<string>& searchPhrase) {
```

b) This function returns the name of the tea (a string) associated with the given tea ID (an integer). Use the *error()* function to throw an error if the tea ID is not found within the map.

```
string getName(Map<string, int>& teaNamesToIDs, int teaID) {
```

c) Return a set of strings as required by the problem description above. Keep in mind you must design your solution to rely on *containsPhrase()* and *getName()*.

```
Set<string> matches(Map<int, Vector<Vector<string>>>& teaDescriptions,
                    Map<string, int>& teaNamesToIDs, Vector<string>& searchPhrase) {
```

This page is intentionally left blank for you to use as scratch paper.

We will not grade anything on this page.

2. Big-O (30 pts) Consider the following functions, both of which determine whether a vector contains all integers from its minimum value through its maximum value with no duplicates and no missing integers along that range:

```
bool containsRunWithSet(Vector<int>& v)
{
    if (v.size() == 0)
    {
        return true;
    }

    int min = v[0];
    int max = v[0];

    Set<int> set;

    for (int i : v)
    {
        if (i < min) min = i;
        if (i > max) max = i;
        set.add(i);
    }

    // If set size == vector size, there must not have been any duplicates.
    // If, in addition, the set size is (max - min + 1), it must contain every
    // integer value on the range min through max. Hooray!
    return set.size() == v.size() && set.size() == (max - min + 1);
}
```

```
bool containsRunWithSort(Vector<int>& v)
{
    v.sort();  // Do not overlook the runtime impact of this line.

    for (int i = 0; i < v.size() - 1; i++)
    {
        // Each successive element should be one greater than the element
        // before it. If not, we either have a duplicate value or a missing
        // integer on the range 'min' through 'max', so we return false.
        if (v[i] + 1 != v[i + 1])
        {
            return false;
        }
    }

    // If we get here, we must have every integer on the range 'min' through
    // 'max' (no duplicates, no missing integers).
    return true;
}
```

This problem continues on the following page.

Please be an **absolute champ** and ensure all your answers below are neatly confined within the given boxes.

a) Using Big-O notation, what is the **worst-case** runtime for *containsRunWithSet()*, assuming it receives a vector with *n* integers, whose minimum integer is *q* and whose maximum integer is *t*?

O( )

b) Using Big-O notation, what is the **best-case** runtime for *containsRunWithSet()*, assuming it receives a vector with *n* integers, whose minimum integer is *q* and whose maximum integer is *t*?

O( )

c) Using Big-O notation, what is the **worst-case** runtime for *containsRunWithSort()*, assuming it receives a vector with *n* integers, whose minimum integer is *q* and whose maximum integer is *t*?

O( )

d) Using Big-O notation, what is the **best-case** runtime for *containsRunWithSort()*, assuming it receives a vector with *n* integers, whose minimum integer is *q* and whose maximum integer is *t*?

O( )

e) In just a few words, what key design flaw does *containsRunWithSort()* have that *containsRunWithSet()* does not? (Note: The answer is unrelated to the correctness of the function's return value or its runtime.)

Next, consider the following function:

```
int foo(int n)
 {
     if (n == 0)
     {
         return 0;
     }
     return n + foo(n – 1) + foo(n – 1);
 }
```

f) Using Big-O notation, what is the runtime for *foo(n)*?

O( )

Consider the following function:

```
int binarySearchParty(Vector<int>& haystack, Vector<int>& needles)
{
     int foundCount = 0;

     for (int thisNeedle : needles)
     {
         // You may assume haystack is passed to binarySearch() by reference and
         // that our binary search function is implemented correctly and efficiently.
         if (binarySearch(haystack, thisNeedle))
         {
             foundCount++;
         }
     }

     return foundCount;
 }
```

g) Using Big-O notation, what is the **best-case** runtime for *binarySearchParty()*, assuming *haystack* contains *k* integers and *needles* contains *q* integers?

O( )

h) Using Big-O notation, what is the **worst-case** runtime for *binarySearchParty()*, assuming *haystack* contains *k* integers and *needles* contains *q* integers?

O( )

i) Give a **sorted** *haystack* vector with **five elements** and a *needles* vector with **five elements** that, when passed to *binarySearchParty()*, will cause the function to incur its **best-case** runtime possible for inputs of that size.

haystack =

needles =

j) Give a **sorted** *haystack* vector with **five elements** and a *needles* vector with **five elements** that, when passed to *binarySearchParty()*, will cause the function to incur its **worst-case** runtime possible for inputs of that size.

haystack =

needles =

This page is intentionally left blank for you to use as scratch paper.

We will not grade anything on this page.

3. Recursion (20 pts)

Write a **recursive** function that takes two vectors, *v1* and *v2*, and returns *true* if they contain the exact same elements, but in reverse order from one another. Otherwise, return *false*. For example:

```
Sample Input Vectors:
v1 = {10, 3, 9, 15, 5}
v2 = {5, 15, 9, 3, 10}
v3 = {9, 1234, 42, 333}
v4 = {333, 42, 1234, 9}
v5 = {}
```

```
Sample Function Calls:
isReverse(v1, v2)  // true
isReverse(v2, v1)  // true
isReverse(v3, v4)  // true
isReverse(v1, v4)  // false
isReverse(v1, v5)  // false
isReverse(v5, v5)  // true
```

In solving this problem, you must abide by the following guidelines and restrictions:

1. You must solve this problem **recursively**.
2. Your function cannot contain any loops.
3. You cannot create any helper functions. All your work must be done in a **single function**.
4. You cannot modify the function signature given below. (You cannot add additional parameters.)
5. Each call to your recursive function must process only a **single element** from each of the given vectors and rely on subsequent recursive calls to handle the rest.
6. Notice that the vectors are passed by **reference**. You should be careful that any time your function returns, the vectors are in the same state they were in when that function was called.
7. You cannot create new copies of the vectors to pass to the function recursively.
8. You will probably have to modify at least one vector in a way that feels inefficient. Do not worry about that. Focus instead on correctness.

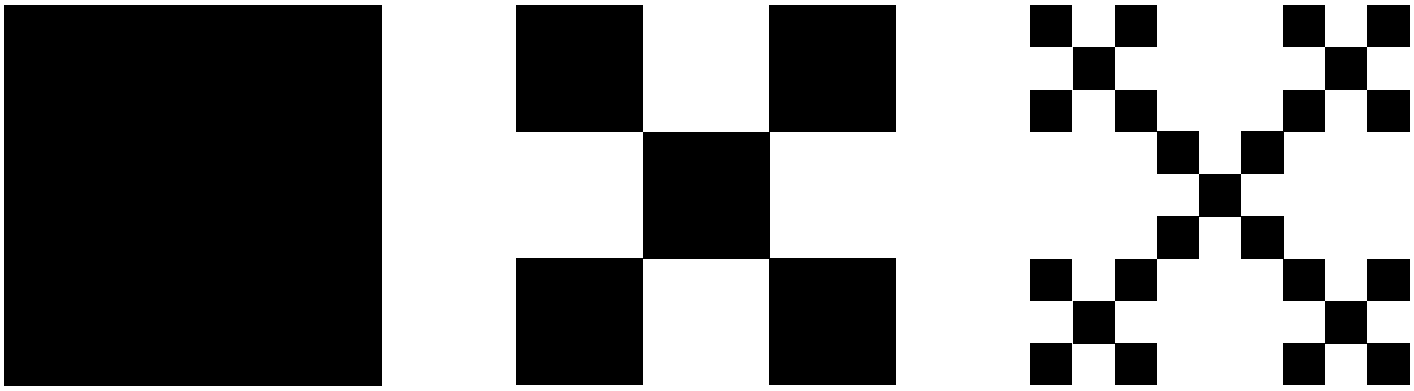
Please write your answer on the following page.

We will not grade any work on this page.

```
bool isReverse(Vector<int>& v1, Vector<int>& v2) {
```

4. Recursive Tracing (10 pts)

Consider the following fractal. Notice that until we hit a base case, each square is effectively divided into a 3x3 grid, and we make five recursive calls to fill in five of the resulting grid's nine squares.

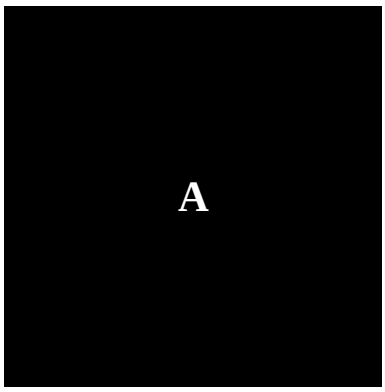


level = 0 *level* = 1 *level* = 2

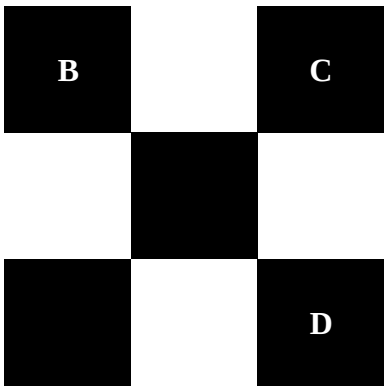
These images were produced with the following function. You can trust the comments in the code. You do not have to reflect carefully on any of the math in order to answer the questions on the following page.

```
void drawXBox(GWindow& w, int level, double topLeftX, double topLeftY, double width)
{
    if (level == 0)
    {
        w.fillRect(topLeftX, topLeftY, width, width);
        return;
    }

    // Width of box for recursive call. Also multiplied by 2 below to calculate
    // width and height 2/3 of the way across and down from this box's start point.
    double mWidth = width / 3.0;

    // Recursive call for top-left portion of fractal.
    drawXBox(w, level - 1, topLeftX, topLeftY, mWidth);

    // Recursive call for top-right portion of fractal.
    drawXBox(w, level - 1, topLeftX + mWidth * 2, topLeftY, mWidth);

    // Recursive call for center portion of fractal.
    drawXBox(w, level - 1, topLeftX + mWidth, topLeftY + mWidth, mWidth);

    // Recursive call for bottom-left portion of fractal.
    drawXBox(w, level - 1, topLeftX, topLeftY + 2 * mWidth, mWidth);

    // Recursive call for bottom-right portion of fractal.
    drawXBox(w, level - 1, topLeftX + 2 * mWidth, topLeftY + 2 * mWidth, mWidth);
}
```

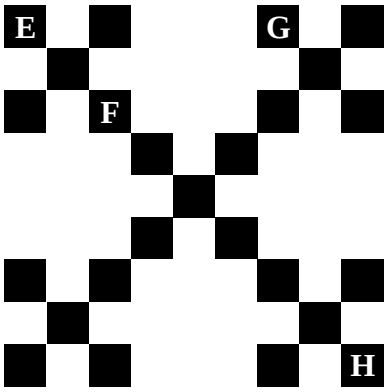
Here are those diagrams again for your reference:



level = 0



level = 1



level = 2

In order to draw the level-two fractal, we of course have to go through the recursive calls depicted for the level-zero and level-one fractals – although rather than drawing a square, those calls will simply make further recursive calls of their own. For example, the very first function call when drawing a level-two fractal is the one depicted by **A** above, although it does not print the square labeled **A**. Instead, that call makes the five recursive calls depicted in the level-one image, and those five calls each make their own five recursive calls.

The goal of this problem is to indicate the order in which certain of these recursive calls are made when drawing a **level-two** fractal, as well as the order in which they return. (Note: You can trust the comments in the code about which box is handled by each recursive call.)

First, let's focus on the order in which the calls are *made*:

- Out of the calls labeled **B**, **C**, **D**, **E**, **F**, **G**, and **H**, which one is made **first**? _____
- Out of the calls labeled **B**, **C**, **D**, **E**, **F**, **G**, and **H**, which one is made **second**? _____
- Out of the calls labeled **B**, **C**, **D**, **E**, **F**, **G**, and **H**, which one is made **third**? _____
- Out of the calls labeled **B**, **C**, **D**, **E**, **F**, **G**, and **H**, which one is made **fourth**? _____
- Out of the calls labeled **B**, **C**, **D**, **E**, **F**, **G**, and **H**, which one is made **last**? _____

Next, let's focus on the order in which these calls *return*:

- Out of the calls labeled **B**, **C**, **D**, **E**, **F**, **G**, and **H**, which one returns **first**? _____
- Out of the calls labeled **B**, **C**, **D**, **E**, **F**, **G**, and **H**, which one returns **second**? _____
- Out of the calls labeled **B**, **C**, **D**, **E**, **F**, **G**, and **H**, which one returns **third**? _____
- Out of the calls labeled **B**, **C**, **D**, **E**, **F**, **G**, and **H**, which one returns **fourth**? _____
- Out of the calls labeled **B**, **C**, **D**, **E**, **F**, **G**, and **H**, which one returns **last**? _____

Fill in each of the blanks above with a single letter.

5. Problem Solving (10 pts)

Write an **iterative** function that takes a single integer parameter, *n*, and prints all sequences of coin flips of length *n*. For example, for *n* = 2, your function should print “HH”, “HT”, “TH”, and “TT”. (As long as you print the correct results, we don’t care how the output is formatted.)

We have seen a recursive version of this function in class. The goal is to now write this iteratively. You can do this! Here are a few hints and restrictions to guide you:

- You can create one or two queues of strings – i.e., *Queue<string>* variables.
- Other than the one or two queues mentioned above, you cannot create any ADTs.
- You cannot write any helper functions. All your work must be done in a single function.
- Hint: You have done something very similar to this already! Think of the way you iterated on and extended the paths you created while working on the DFS and BFS maze solutions.

The function signature is:

```
void printCoinFlips(int n) {
```