Practice Midterm 3

CS106B, Fall 2023

_____

(Print name legibly)

_____

(SUID **number**)

Exam Instructions

There are **4** questions worth a total of **70** 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.

Problem 1: C++ Fundamentals and ADTs (17 points)

A *concordance* is a type of index that lists the words in a document and associates each word with the position(s) at which it appears within the document. Positions are zero-indexed; the first word is at position 0, the second at 1, and so on. A concordance can be modeled in C++ using a `Map<string, Set<int>>` where the words are keys and the associated value is a set of positions. All words in the concordance are stored in lowercase.

A document contains the text:

```
One Fish two fish RED FISH blue fish two red fish
```

List the entries in the concordance built from this document:

The `buildConcordance` function reads the document text from a file and builds a concordance. The function is started below with correct code to open the file and a loop to read the words one by one. Complete the function by adding the code to update the appropriate concordance entries.

```
void buildConcordance(string filename, Map<string, Set<int>>& conc)
{
    int pos = 0;
    ifstream in;
    if (openFile(in, filename)) {
        string word;
        while (in >> word) { // loop read next word from file until done

        }
    }
}
```

A *phrase* is a sequence of words. The function **findPhrase** searches a concordance to find a sequence of words in the document that matches a given phrase. It returns the start position of the first phrase match or -1 if the phrase is not found.

Here are some valid test cases for the concordance on the previous page:

```
EXPECT_EQUAL(findPhrase("red", concordance), 4);
EXPECT_EQUAL(findPhrase("Fish Blue Fish", concordance), 5);
EXPECT_EQUAL(findPhrase("red blue", concordance), -1);
```

Write the function **findPhrase**. The **phrase** argument is a string consisting of a sequence of words separated by spaces. The **stringSplit** function can be used to divide a string into words. Words should be matched case-insensitively.

```
int findPhrase(string phrase, Map<string,Set<int>>& conc)
```

Problem 2: Code study: ADTs and Big-O (15 points)

The *echo* operation accesses elements in a collection and adds the element's "echo" (value divided by two). Below are four versions of echo; one each for Vector, Stack, Queue, and Set. The provided code compiles and runs without error but may or may not work as intended.

Fill in the boxes below with the Big-O runtime for each echo function in terms of N where N is the size of the collection.

```
void echoVector(Vector<int>& v)
{
    for (int i = v.size() - 1; i >= 0; i--) {
        int cur = v[i];
        v.insert(i, cur/2);
     }
}
```

O( )

```
void echoStack(Stack<int>& s)
{
    for (int i = s.size() - 1; i >= 0; i--) {
        int cur = s.pop();
        s.push(cur);
        s.push(cur/2);
    }
}
```

O( )

```
void echoQueue(Queue<int>& q)
{
    for (int i = q.size() - 1; i >= 0; i--) {
        int cur = q.dequeue();
        q.enqueue(cur);
        q.enqueue(cur/2);
    }
}
```

O( )

```
void echoSet(Set<int>& set)
{
    Set<int> echoes;
    for (int cur: set) {
        echoes.add(cur/2);
    }
    set.unionWith(echoes);
}
```

O( )

Fill in the boxes below to show the contents of each collection **after** the call to echo.

```
Vector<int> v = {3, 5, 6};
echoVector(v);
```

v =

```
Stack<int> s = {3, 5, 6};
echoStack(s);
```

s =

```
Queue<int> q = {3, 5, 6};
echoQueue(q);
```

q =

```
Set<int> set = {3, 5, 6};
echoSet(set);
```

set =

*Reminder: **Stack** elements are listed in order **bottom to top**. **Queue** elements listed in order **front to back**.*

Your co-worker is curious why the loop in `echoQueue` iterates backwards; it seems especially odd given that the value of index variable `i` is never used. They edit the code to iterate over the same indexes in forward order. The revised loop header is now:

```
for (int i = 0; i < q.size(); i++) {
```

Explain the consequence this change will have on the behavior/output of `echoQueue`.

Your boss suggests rewriting `echoSet` in the more direct form shown below:

```
void echoSet(Set<int>& set)
{
    for (int cur: set) {
        set.add(cur/2);
    }
}
```

You test the above version of `echoSet` and it raises a runtime error. What is the problem?

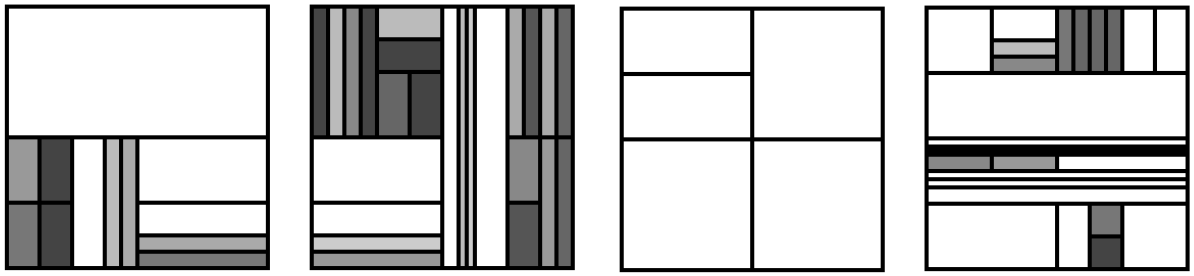
Problem 3: Recursion (18 points)

A Mondrian rectangle is a self-similar pattern that is defined recursively.

- A simple Mondrian is a filled grayscale rectangle. The simple case applies to a Mondrian whose area is less than 500 pixels.
- A larger Mondrian can take one of three possible forms:
  1) a filled white rectangle (no split)
  2) two half-width Mondrian rectangles placed side-by-side (vertical split)
  3) two half-height Mondrian rectangles placed top-to-bottom (horizontal split)

A random choice of which split option (1, 2, or 3) is made for each level of a Mondrian fractal.

Below are four Mondrian rectangles of size width and height 100 (total area = 10,000 pixels). Because splits and grayscale colors are randomly chosen, each run produces a different result. The label beneath each Mondrian is the percentage of the fractal's total area that is filled in white.



75% white | 37.5% white | 100% white | 81.25% white

You are to write the recursive fractal function

```
double drawMondrianRect(double x, double y, double w, double h)
```

The parameters to `drawMondrianRect` specify the rectangle in which to draw. The lower-left corner is at `(x,y)` and the rectangle size is `h` pixels tall by `w` pixels wide. The total area is `w * h`. The function draws a Mondrian fractal in the specified rectangle and returns the percentage of the fractal's total area that was filled in white.

Additional requirements

- Call `randomInteger(int low, int high)` to choose a random value from the range low through high (inclusive).
- Call `drawRect(double x, double y, double w, double h, string color)` to draw a rectangle filled with a specified color. The color parameter is either `"white"` or `"grayscale"` (for a random gray value).
- The return value from `drawMondrianRect` is a percentage, expressed as a number from 0 to 100, indicating the ratio of the area of the white rectangles to the fractal's total area.
- You will need a **helper/wrapper** function. Use the recursive helper to draw the fractal and sum the white areas. In the wrapper, compute the percentage. (Hint: this strategy is similar to compute power indexes!)

```
double drawMondrianRect(double x, double y, double w, double h)
```

This page intentionally left blank. You may use this space for scratch work.
It will not be graded unless you write a redirect from the original answer area to here.

Problem 4: Recursive backtracking (20 points)
You have a $100 gift card good for a one-outing shopping spree and want to assemble a basket of items that spends every cent to avoid forfeiting any unspent value. This is a perfect job for your recursive backtracking skills!

```
bool spendAll(double amount, Vector<itemT>& inventory)
```

You are to write the `spendAll` function which recursively forms baskets of items from the inventory to search for a basket that exactly sums to the full amount.

The store inventory is stored in a vector of `itemT`, a struct for the item name, price, and count.

```
struct itemT {
    string name;
    double price;
    int count; // count of item available, 0 if out of stock
};
```

A basket can contain 0, 1 or up to N of an item where N is the available count of the item.

If a basket is found that exactly spends the full amount, the counts of purchased items are decremented from the inventory and `true` is returned. If no such basket is found, the inventory is unchanged and the function returns `false`.

Additional requirements

- Your algorithm must be **recursive** and must use **backtracking** to generate the result.
- You must **limit the recursive exploration** by stopping at the first success and pruning all dead-end paths as soon as you can detect they are not viable.
- You should make **efficient choices** in handling data structures within your recursive calls: make no unnecessary copies, no expensive edits to the inventory vector, and use no additional data structures. (Rule of thumb: on par with count critical votes 24 items in 5 seconds).
- The function returns a true/false result of whether the entire amount was spent. **Do not print, store, or return** the basket of items.
- You will need a **helper/wrapper** function.

Example

```
Vector<itemT> inventory = {{"apple", 0.50, 1}, {"banana", 0.75, 5}};
```

`spendAll(1, inventory)` returns `false` (no basket totals to exactly $1)
`spendAll(3.50, inventory)` returns `true` and updates inventory to
`{{"apple", 0.50, 0}, {"banana", 0.75, 1}}`

Below we provide the code for the function **printTotalSpent** which prints the total cost for each possible basket of items. This is a different task than **spendAll** but has some similarities.

```
void printTotalSpent(double soFar, Vector<itemT>& inventory)
{
    if (inventory.isEmpty()) {
        cout << "Total spent: " << soFar << endl;
        return;
    }
    itemT item = inventory.remove(0);
    printTotalSpent(soFar, inventory);
    printTotalSpent(soFar + item.price, inventory);
    inventory.insert(0, item);
}
```

Review the above code to determine what you can adopt from it. Your function will take a similar approach, but take note of key differences such as:

- **printTotalSpent** prints the total for every basket. Your function must **not print** anything.
- **printTotalSpent** considers adding either zero or one of an item to the basket. Your function tries adding **zero up to N** where N is the available count of the item.
- **printTotalSpent** does not return a result. Your function returns **true or false** to indicates whether a successful basket was found.
- **printTotalSpent** does a fully exhaustive search of all possible baskets. Your function must **stop exploration** at the first successful basket and **prune** dead end paths.
- **printTotalSpent** runs very slowly due to suboptimal choices in managing the vector in the recursive calls. Your function must use **efficient handling** of data structures to avoid unnecessary slowdown.

Please write your answer on the following page.

```
bool spendAll(double amount, Vector<itemT>& inventory)
```

This page intentionally left blank. You may use this space for scratch work.
It will not be graded unless you write a redirect from the original answer area to here.