Practice Midterm 4

CS106B, Winter 2024

_____

(Print name legibly)

_____

(SUID **number**)

Exam Instructions

There are **4** questions worth a total of **80** 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 (20 pts)

Moira's Teashop of Wonderment and Whimsy serves phenomenal teas with unusual names. They have been tracking customer purchases and want to create a system to offer customers recommendations for teas they might enjoy but haven't tried yet, based on the teas that customers with similar tastes have enjoyed.

Customer information is stored in a struct:

```
struct infoT
{
   string name;
   Set<string> favorites;
};
```

Here is some sample **`infoT`** data for your consideration:

name: "Aabria",
favorites: {
"Tea of Jollity",
"Tea of Jubilation",
"Cherry Jubilee Tea",
"Tea of Wisdom"
}
	name: "Emily",
	favorites: {
	"Tea of Jollity",
	"Tea of Jubilation",
	"Tea of Wonderment",
	"Tea of Whimsy",
	"Tea of Lucid Dreams"
	}
		name: "Lou",
		favorites: {
		"Tea of Prosperity",
		"Tea of Lucid Dream",
		"Tea of Jollity",
		"Tea of Jubilation",
		"Tea of Whimsy"
		}
			name: "Siobhan",
			favorites: {
			"Tea of Prosperity",
			"Cherry Jubilee Tea"
			}

(a) Write a function that takes two **`infoT`** structs, **`target`** and **`source`**, and determines whether there is enough overlap in their favorite teas to use **`source`** as a source of new tea recommendations for **`target`**. If at least half of **`target`**'s favorite teas are in **`source`**'s set of favorite teas, return a set of all of **`source`**'s favorite teas that are **not** in **`target`**'s set. Otherwise, if there is not enough intersection between their favorite teas, return an empty set. For example:

If Aabria is **`target`** and Emily is **`source`**, return: {"Tea of Wonderment", "Tea of Whimsy", "Tea of Lucid Dreams"}

If Emily is **`target`** and Aabria is **`source`**, return: {}

↳ *Not enough overlap. Less than ½ Emily's favorites were in Aabria's list.*

The function signature is:

```
Set<string> teaFromSource(infoT& target, infoT& source)
```

(b) Write a function that takes a vector of **`infoT`** structs (**`theTea`**) and returns a **`Map<string, Set<string>>`** mapping the name of each customer to the tea recommendations produced by comparing them to all other customers in the vector. You may assume each customer has a unique name. You must use your **`teaFromSource()`** function when solving this problem. For the customers above, this function would return the following map:

```
{
    "Aabria" :   {"Tea of Wonderment", "Tea of Whimsy", "Tea of Lucid Dreams", "Tea of Prosperity"},
    "Emily":     {"Tea of Prosperity"},
    "Lou":       {"Tea of Wonderment"},
    "Siobhan":   {"Tea of Jollity", "Tea of Jubilation", "Tea of Wisdom", "Tea of Lucid Dreams", "Tea of Whimsy"}
}
```

The function signature is:

```
Map<string, Set<string>> getRecommendations(Vector<infoT>& theTea)
```

The function descriptions and struct definition from the previous page are repeated here for your convenience.

```
struct infoT
{
   string name;
   Set<string> favorites;
};
```

(a) Write a function that takes two **`infoT`** structs, **`target`** and **`source`**, and determines whether there is enough overlap in their favorite teas to use **`source`** as a source of new tea recommendations for **`target`**. If at least half of **`target`**'s favorite teas are in **`source`**'s set of favorite teas, return a set of all of **`source`**'s favorite teas that are not in **`target`**'s set. Otherwise, if there is not enough intersection between their favorite teas, return an empty set.

```
Set<string> teaFromSource(infoT& target, infoT& source) {
```

(b) Write a function that takes a vector of **`infoT`** structs (**`theTea`**) and returns a **`Map<string, Set<string>>`** mapping the name of each customer to the tea recommendations produced by comparing them to all other customers in the vector. You may assume each customer has a unique name. You must use your **`teaFromSource()`** function when solving this problem.

```
Map<string, Set<string>> getRecommendations(Vector<infoT>& theTea) {
```

2. Big-O (20 pts)

Give the Big-O runtimes for the following functions. Use *n* to represent the vector's size.

```
bool indexIsValid(Vector<int> v, int index) {
   // Recall that v.size() is O(1).
   return index >= 0 && index < v.size();
}
```

Best-case runtime: O( )

Worst-case runtime: O( )

```
// There is a subtle difference between this
// function and the one above. Look carefully!

bool indexIsValid(Vector<int>& v, int index) {
   // Recall that v.size() is O(1).
   return index >= 0 && index < v.size();
}
```

Best-case runtime: O( )

Worst-case runtime: O( )

```
int countAndDeplete(Vector<int>& v, int target) {
   int count = 0;
   while (!v.isEmpty()) {
      if (v.remove( ??? ) == target) {
         count++;
      }
   }
   return count;
}
```

What is the runtime if we replace the `???` in this code with `0` (zero)? O( )

What is the runtime if we replace the `???` in this code with `v.size() - 1`? O( )

```
// Assume the vector is sorted and we have an
// efficient implementation of binary search that
// returns a boolean to indicate whether target is
// found.
bool contains(Vector<int>& v, int target) {
   return binarySearch(v, target);
}
```

Best-case runtime: O( )

Worst-case runtime: O( )

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)

Problem Overview

In this problem, you will do two things:

1. Read about postfix expressions. Spend some time understanding what they are and how they work.
2. Write a recursive function to determine whether a given postfix expression is valid.

Postfix Expressions

A postfix expression is an arithmetic expression in which operators (`+`, `-`, `*`, `/`) are listed after their operands (the values upon which they operate). The infix expression `a + b` is written in postfix as `a b +`. The neat thing about postfix expressions is that they can be processed from left to right without having to worry about operator precedence. Because of that, calculator applications often convert infix expressions to postfix behind the scenes for evaluation.

Here's the process for evaluating a postfix expression:

1. Look through the expression from left to right until you encounter the first operator.
2. Apply the operator to the two values to its left. The first of those values is its left-hand operand. The other is its right-hand operand. For example, `a b /` corresponds to `a/b` in infix (not `b/a`).
3. Replace all three tokens (the operator and its two operands) with the result of that operation.
4. Go back to Step 1 and repeat until the expression has been processed.

For example, the evaluation of the postfix expression `3 5 2 + + 8 - 3 4 * +` is as follows:

1. Scanning from left to right, the first operator we encounter is `+`. We replace `5 2 +` with `7` (since `5 + 2 = 7`), and our expression becomes: `3 7 + 8 - 3 4 * +`
2. The next operator (`+`) gives us: `3 + 7 = 10`. Our expression becomes: `10 8 - 3 4 * +`
3. The next operator (`-`) gives us: `10 - 8 = 2`. Our expression becomes: `2 3 4 * +`
4. The next operator (`*`) gives us: `3 * 4 = 12`. Our expression becomes: `2 12 +`
5. The next operator (`+`) gives us: `2 + 12 = 14`. Our expression becomes: `14`
6. There are no more operators, so we finish. The result is `14`.

Your Task

Write a recursive function that simply determines whether a given postfix expression is valid. Do **not** evaluate the numeric value of the expression. The expression is valid if all these conditions are met:

1. As we process the expression from left to right, every time we see an operator, there are always *at least* two remaining value to its left. Step 3 above shows that we don't always have *exactly* two values to the left of an operator. The `*` operator has three values to its left. The expression is still valid.
2. At the end of evaluation, we have exactly one value remaining and no operators. If we have more than one value left and no operators, or if the expression is empty initially, it is not valid.
3. It contains only operators and values – no other tokens.

Function Specifications

Your function should return **`true`** if the expression it receives is valid postfix, **`false`** otherwise. To make your life easier, the expression is passed to your function as a vector of strings. For example (using the expression from the previous page): `isValid({"3", "5", "2", "+", "+", "8", "-", "3", "4", "*", "+"})`

The function signature is:

```
// Return true if the given expression is valid postfix, false otherwise.
bool isValid(Vector<string> expr)
```

You can assume the following functions already exist and can be called from `isValid()`:

```
// Returns true if s is an operator (+, -, *, /), false otherwise.
bool isOperator(string s);

// Returns true if s is a numeric value, false otherwise.
bool isNum(string s);
```

Ground Rules

Please follow these ground rules when writing your function. These are here to help guide you!

1. You must solve this problem **recursively**.
2. You can remove elements from the vector, but you cannot add or modify any elements in the vector. If you find yourself wanting to add or modify vector elements, you are probably trying to evaluate the postfix expression, which is not the goal of this problem.
3. If you remove elements from the vector, you don't have to add them back later. The vector is passed to this function by value initially, so we're operating on a copy. Destroying it isn't a problem.
4. You will almost certainly need to write a recursive helper function.
5. For the sake of efficiency, pass the vector to your recursive helper function by reference.
6. **Do not** create any data structures to pass to your function recursively!

Test Cases

Here are some additional test cases to help you on your way:

```
isValid({ "20" })  // True. No operators. Finishes with one value in expression.
isValid({ "20", "50" })  // False. No operator. Finishes with two values in expression.
isValid({ "20", "+", "50" })  // False. + operator does not have two values to its left.
isValid({ "+", "20" })  // False. + operator does not have two values to its left.
isValid({ "20", "50", "+", "+", "10" })  // False. Second + won't have two vals to its left.
isValid({ "20", "+", "50", "+",  "30"})  // False. First + won't have two vals to its left.
isValid({ "1", "2", "+", "3", "+", "4", "+", "5", "+", "6", "+" })  // True
isValid({ "1", "2", "3", "4", "5", "6", "+", "+", "+", "+", "+" })  // True
isValid({ "1", "2", "3", "+", "+", "4", "+", "5", "6", "+", "+" })  // True
isValid({ "1", "2", "3", "4", "5", "6", "+", "+", "+", "+", "+", "+" })  // False
```

BIG HINT: Keep track of the number of values we have encountered that have not yet been operated upon. What happens to that count when we see a value? What happens to it when we see an operator?

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

We will not grade anything on this page.

Reminders:

1. You must solve this problem **recursively**.
2. There are `isOperator()` and `isNum()` functions you can call! See pg. 7.

```
bool isValid(Vector<string> expr) {
    // code me



}
// Finish the recursive helper function's signature and write the function.
bool isValidHelper(Vector<string>& expr,
```

4. Recursive Backtracking (20 pts)

Write a recursive *sumWords()* function that counts up and prints all valid English words whose characters' ASCII values add up to a given target integer. For example, if *target* is 650, one of the many strings your function should print is "wombat", since 'w' + 'o' + 'm' + 'b' + 'a' + 't' = 650.

- Your algorithm must be recursive and must use backtracking techniques to generate its results.
- The function must print all solutions to the screen **and** return the number of solutions it finds.
- Focus only on strings with lowercase alphabetic letters – no uppercase or non-alpha characters.
- You may assume we have a lexicon with valid English words. See function signature on next page.
- You must not create any auxiliary data structures!
- You will need to write a helper function.
- You should only explore sequences that could potentially lead to valid results. As soon as it becomes clear that a sequence you have generated cannot lead to any valid results, stop exploring that dead-end path.
    - ***Hint:*** *There are at least two situations that should cause us to give up early and stop making recursive calls without having found a valid solution on a given path.*
- You should find a way to avoid repeated addition as you build up a particular string. For example, if you have added up the ASCII values for all the characters in the string "teapot" and then make a recursive call on the string "teapots", you should not have to add up the ASCII values for the "teapot" portion of "teapots" from scratch.

For your reference, here is a *printCombos()* function that uses recursion to print all possible letter sequences of a given length. Read over this code as a starting point. You can borrow code and structure from *printCombos()* when writing *sumWords()*.

```
void printCombos(int length, string soFar)
{
    if (soFar.length() == length)
    {
        cout << soFar << endl;
    }
    else
    {
        for (char ch = 'a'; ch <= 'z'; ch++)
        {
            printCombos(length, soFar + ch);
        }
    }
}

// wrapper function
void printCombos(int length)
{
    printCombos(length, "");
}
```

```
// You may assume the lexicon comes pre-loaded with a list of English words.
int sumWords(int target, Lexicon& lex) {

    // code me



}

// Recursive helper function goes here:
```