Practice Midterm 4

CS106B

_____

(Print name legibly)

_____

(SUID **number**)

Exam Instructions

There are **4** 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. Big-O (30 pts)

```
// Takes a sorted vector (v) and an integer (k) and returns true if the element at
// index v.size() / 2 (approximately the middle element of the vector) occurs at least
// k times (that is, k times or more). Otherwise, the function returns false.
bool atLeastKCopies(Vector<int>& v, int k)
{
   int mid = v.size() / 2;
   int target = v[mid];
   int count = 1;

   // Count how many times target occurs at indices 0 through (mid – 1).
   int i = mid – 1;
   while (i >= 0 && v[i] == target) {
      count++;
      i--;
   }

   // Count how many times target occurs at indices (mid + 1) through (n – 1).
   i = mid + 1;
   while (i < v.size() && v[i] == target) {
      count++;
      i++;
   }

   return (count >= k);
}
```

Consider the function above, and then answer the following questions. For this problem, use ***n*** if you need to refer to the size of the vector in any of your responses.

a) Using Big-O notation, give the **best-** and **worst-case** runtimes for the *atLeastKCopies()* function.

O( )	O( )
Best-Case Runtime	Worst-Case Runtime

b) Give a **sorted** vector (*v*) and an integer (*k*) that would cause this function to encounter its **best-case** runtime.

v:

0	1	2	3	4	5	6	7	8	9

k: ____

c) Give a **sorted** vector (*v*) and an integer (*k*) that would cause this function to encounter its **worst-case** runtime.

v:

0	1	2	3	4	5	6	7	8	9

k: ____

d) What bug does this function suffer from? Give an input that causes the bug and indicate what the undesired result is. Your response must be **10 words or fewer** in order to receive credit. (The bug is **not** an obscure, minor syntax thing.)

```
// Takes a sorted vector (v) and an integer (k) and returns true if the elements in
// the middle of the vector occurs at least k times, false otherwise.
bool atLeastKCopies(Vector<int>& v, int k)
{
   int mid = v.size() / 2;
   int target = v[mid];
   int count = 1;

   // Count how many times target occurs at indices 0 through (mid – 1).
   int i = mid – 1;
   while (i >= 0 && v[i] == target) {   // Note: This short-circuits if i < 0.
      count++;
      i--;

      if (count >= k)   // new!
      {                 // new!
         return true;   // new!
      }                 // new!
   }

   // Count how many times target occurs at indices (mid + 1) through (n – 1).
   i = mid + 1;
   while (i < v.size() && v[i] == target) {   // Note: This short-circuits
      count++;                                //       if i >= v.size().
      i++;

      if (count >= k)   // new!
      {                 // new!
         return true;   // new!
      }                 // new!
   }

   return false;        // modified!
}
```

Consider the function above, which is a **<u>modified version</u>** of the function from the previous page, and then answer the following questions. Again, use ***n*** if you need to refer to the size of the vector in any of your responses.

e) If we assume ***k*** < ***n***, what are the **<u>best-</u>** and **<u>worst-case</u>** runtimes for this modified version of the function?

O( )	O( )
Best-Case Runtime	Worst-Case Runtime

f) If we assume ***n*** < ***k***, what are the **<u>best-</u>** and **<u>worst-case</u>** runtimes for this modified version of the function?

O( )	O( )
Best-Case Runtime	Worst-Case Runtime

This question is continued on the following page.

g) The **modified version** of the function that appears on the previous page introduces a **new bug** that was not present in the original function on the page before that. (Note that this new bug is problematic whether or not the bug from the previous version has been fixed. Also, like before, this bug is not an obscure, minor sytnactic problem that would require careful scrutiny of every semicolon and >= operator.) Give a **sorted** vector of integers (*v*) and an integer (*k*) that trigger this new bug. Indicate what the function returns for those inputs, and indicate what it *should* return if the function were implemented correctly.

v: [ ] *k*: [ ]

(Write your vector here. You can use curly brace notation.)

Given the inputs *v* and *k* specified above, this version of the function would return: [ ]

true/false

Given the inputs *v* and *k* specified above, a **correct** version of the function would return: [ ]

true/false

Please be an **absolute champ** and ensure all your answers for all parts of this problem are neatly confined within the given boxes.

We will not grade anything written in the empty space above.

We will not grade anything written outside the boxes on this page.

This problem is continued on the following page.

2. Problem Solving with ADTs (30 pts)

In class, we wrote a program that tracked word frequencies in a map of type *Map<string, int>*. We fed the map a word (a string that served as a *key*) and it produced the frequency (an integer *value*) with which that word occurred in some input.

In this problem, you will write a function that produces an **inverted** frequency map; it will map a frequency (an integer *key*) to an ADT containing all the strings we have seen with that particular frequency. It will be up to you to choose an ADT that makes the most sense to place within your map for this task.

Your function will take as its input a vector of strings. Each string will contain space-separated words. You can assume the words are already cleaned: they will contain lowercase letters only (no uppercase letters or non-alphabetic characters). You are to build an inverted frequency-to-container-of-strings map from that input vector.

For example, suppose your function received the following vector of strings:

```
{
    “roses are red”,
    “butterflies are beautiful”,
    “but the butterflies revolted”,
    “they were done being dutiful”
}
```

The frequencies of all the words in that vector are as follows:

string	frequency
are	2
beautiful	1
being	1
but	1
butterflies	2
done	1
dutiful	1
red	1
revolted	1
roses	1
the	1
they	1
were	1

Accordingly, the map you create from that input should contain exactly **<u>two</u>** entries:

- an entry mapping the integer 1 to an ADT containing the strings “beautiful”, “being”, “but”, “done”, “dutiful”, “red”, “revolted”, “roses”, “the”, “they”, and “were” (not necessarily in that order).
- an entry mapping the integer 2 to an ADT containing “are” and “butterflies” (not necessarily in that order).

This problem is continued on the following page.

Specifically, you must write the following functions:

- ***Map***<***int***, **ADT** <***string***>> *buildInvertedMap*(***Vector***<***string***>& *lines*)
  - Create and return a map as described above.
  - When you write this function, replace **ADT** with an appropriate ADT.
- ***void*** *incrementFrequency*(***Map***<***int***, **ADT** <***string***>>& *map*, ***string*** *word*)
  - Update *map* to reflect an additional occurrence of *word*.
    - For example, suppose 1 is mapped to an ADT containing “butterfly” and “wings”, and 2 is mapped to an ADT containing “wishes”. After calling *incrementFrequency(map, “wings”)*, we should end up having 1 mapped to an ADT containing just “butterfly” and 2 mapped to an ADT containing both “wishes” and “wings”.
  - You may assume that *word* will only occur in the map at most once. There will not be multiple frequencies mapping to *word*.
  - If *word* occurs in the map, you will need to remove it from one nested ADT and add it to a different one.
  - If an integer ends up mapping to an empty ADT as a result of this function call, remove that integer from the map altogether.
  - If *word* does not occur in the map, this is the first time it has been encountered, and so you must add it to the ADT associated with the integer 1.
  - When you write this function, replace **ADT** with an appropriate ADT.

Your task is to complete these functions following these key restrictions:

- Other than the functions listed above, please do not write any helper functions.
- You must call the *incrementFrequency()* function from *buildInvertedMap()* as appropriate.
- Choose an appropriate ADT for the map’s second type parameter.
- Stick to ADTs we have covered in class. Do not use features of C++ that we have not covered in class.
- Make appropriate use of ADTs and library functions. Do not reinvent the wheel.
- Do not leave any empty entries in your map. If you update your map in such a way that an integer ends up mapping to an empty ADT, simply remove that integer from the map altogether.
- You may only loop through the words in the *lines* vector once. You may not make repeated passes over the *lines* vector or any of the individual strings it contains.
- **Do not modify the function signatures** given above (other than to inject an appropriate ADT).
- **Super important restriction:** In creating this map, you **cannot** create a *Map<string, int>* to track the string frequencies and then simply invert it, and you cannot create any auxiliary ADTs other than the ones inside your map and any vectors you create to assist in looping through the strings in the input to this function. This restriction is in place so that we can more fully assess everyone’s comfort in manipulating nested ADTs.

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

We will not grade anything on this page.

There is space for your response to Question #2 on the following page.

Both functions for Question #2 must fit on this page. Please do not overflow onto another page. If you need scratch space before writing your final solution here, please take advantage of the blank pages provided elsewhere on the exam.

3. Recursive Coding (20 pts)

Write a recursive function that takes a vector as its only argument and adds to each element the sum of all previous elements in the vector. For example, suppose we pass the following vector to our function by reference:

```
{60, 12, 33, 85, 62}
```

After calling this function, our vector would become:

```
{60, 72, 105, 190, 252}
```

Note that to 12, we added the sum of all previous elements (60), giving us 60 + 12 = 72. To 33, we added the sum of all previous elements (60 and 12), giving us 60 + 12 + 33 = 105. To 85, we added the sum of all previous elements (60, 12, and 33), giving us 60 + 12 + 33 + 85 = 190. And so on.

The function signature for this problem is as follows:

```
void cascade(Vector<int>& v)
```

Write your solution on the following page, with the following restrictions:

Please read all of the following requirements carefully.

- Your function must be **recursive** to earn credit. Your function cannot have any loops.
- You **must** use the provided function signature for the initial call to this function.
- Regarding wrapper and helper functions:
  - You may use the provided function signature as a wrapper function if needed.
  - You can only write a helper function if you are using the provided function signature as a wrapper to that helper function. Beyond that, **no other helper functions** are permitted.
  - Don't worry about the order of the function definitions (i.e., if you write a helper function, it can go below your wrapper function; you don't need a functional prototype).
  - If you write a helper function, you can only add **one** additional parameter beyond the vector, and it must be a **pass-by-value** parameter (not pass-by-reference).
- You cannot create any new ADTs. You cannot create any copies of the vector passed to this function.
- Do not use features of C++ that we have not covered in class.
- Some of the solutions we have in mind for this problem might require you to modify the contents of the vector in a way that feels uncomfortably inefficient. For this problem, that's okay; you won't be penalized for inefficient vector operations as long as you meet all the other requirements on this page.

- **Super important restriction:** Each call to your function can only ever access a single element within the vector; you **cannot** peek at indices earlier or later in the vector, or access multiple/different elements. In other words, if your function has a line of code that accesses *v*[*i*], it cannot contain a line of code elsewhere that would access some other cell in *v*.

There is space for you to write your function(s) on the following page.

Your code for Question #3 must fit on this page. Please do not overflow onto another page. If you need scratch space before writing your final solution here, please take advantage of the blank pages provided elsewhere on the exam.

```
void cascade(Vector<int>& v) {
```

4. Recursive Thinking (20 pts)

Here is the first recursive permutations function we encountered in class this quarter:

```
void permute(string soFar, string rest) {
   if (rest == "") {
      cout << soFar << endl;
      return;
   }
   for (int i = 0; i < rest.size(); i++) {
      char thisOne = rest[i];

      // the following builds a copy of 'rest' with rest[i] removed
      string subby = rest.substr(0, i) + rest.substr(i + 1);
      permute(soFar + thisOne, subby);
   }
}

void permute(string str) {
   permute("", str);
}
```

Suppose we call ***permute("BOAST")***. After contemplating the function above and considering the order in which it prints its results, answer the questions below. Here are some tips:

- This function does *not* produce strings in the same order as the one that used a *k* variable, so be sure you're not thinking of that version as you solve this problem.
- The goal is not to list all permutations in order to solve this problem (there are too many for that to be an effective use of your time!), but rather to consider how this function works at a higher level.

a) What is the first string this function would print?

b) What is the last string this function would print?

c) Before this function prints "STOAB", would it print **all** the permutations beginning with 'O', **some** (but **not all**) of the permutations beginning with 'O', or **none** of the permutations beginning with 'O'? (Check exactly one box below.)

☐ all ☐ some (but not all) ☐ none

d) What is the first permutation this function would print that begins with ‘A’?

e) Consider the first permutation this function would print that begins with ‘A’. Before it prints that string, would it print **all** of the permutations that begin with ‘T’, **some** (but **not all**) of the permutations that begin with ‘T’, or **none** of the permutations that begin with ‘T’?

☐ all ☐ some (but not all) ☐ none

f) Again, consider the first permutation this function would print that begins with ‘A’. Before it prints that string, would it print **all** of the permutations that begin with ‘S’, **some** (but **not all**) of the permutations that begin with ‘S’, or **none** of the permutations that begin with ‘S’?

☐ all ☐ some (but not all) ☐ none

g) What are all the permutations this function would print that contain “SOBA” as a substring? (Note that we would **not** say that “SOBA” is a substring of “SOTBA”. The characters in “SOBA” must appear consecutively, uninterrupted by any other characters, in order for “SOBA” to be considered a substring.) List **all** permutations that contain “SOBA” as a substring.

h) Consider the point at which this function prints a permutation containing “SOBA” as a substring for the **first time**. At that point, would it already have printed **all** the permutations containing “BOA” as a substring, **some** (but **not all**) of those permutations, or **none** of those permutations?

☐ all ☐ some (but not all) ☐ none

i) Consider the point at which this function prints a permutation containing “SOBA” as a substring for the **last time**. At that point, would it already have printed **all** the permutations containing “BOA” as a substring, **some** (but **not all**) of those permutations, or **none** of those permutations?

☐ all ☐ some (but not all) ☐ none

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

We will not grade anything on this page.