

CS106B Midterm 1

Name:	SUID (numeric):
-------	-----------------

Instructions:

- **Stay within designated answer boxes**, or graders may not see your response on Gradescope. If you require extra space, write “continued” in the answer box, and then use designated scratch space somewhere in the exam. **No additional scratch paper.**
- **The time limit is strict.** When time is called, immediately drop your pencil/pen, and proceed to turn in your exam. No writing.
- For coding problems, **you may assume #include and function prototypes** already exist for you as needed.
- **Stanford Honor Code:** 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 on this examination.
- **Permitted and unpermitted aid:** You may have one double-sided, 8.5” × 11” page of notes. No other aids. **No electronics including laptops, phones, smart glasses/watches or earbuds within reach or sight.** Other than course staff, no consulting with other humans or AI in any way during the exam period.
- **Proctoring:** Proctors will be present in the exam room. If you have any questions or concerns during this exam, please speak with a proctor. At some point during the exam, proctors may take a few broad photos of the room, for later seating chart verification. Backpacks and phones must be left at the front of the room or with proctors. Proctors will verify your photo ID when you turn in your exam.
- If you need a **break, (e.g., bathroom)**, first give your exam to a proctor, who will log your break.

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:

1. ADTs: Stacks and Queues (15 points)

Consider the following code, with the input **value of v as shown in the first cout**. As a reminder for you, below is what the Stanford Library Documentation webpages say about the format when you `cout <<` these structures, and we have marked the answer boxes for you accordingly.

- **Queue**: “The elements are listed left-to-right **from the front of the queue to the back**, such as {value1, value2, value3}.”
- **Stack**: “The output is in the form {value1, value2, value3} where elements are listed left-to-right **from the bottom of the stack to the top**.”
- **Vector**: “Elements are listed **in order of ascending index**.”

```
void collectionMystery(Vector<int>& v) {
    int N = v.size();          // For use in Problem 2 (Big-O).
    cout << v << endl;        // For use in Problem 1, output is {5, 4, 3, 1}

    Stack<int> s;
    for (int i = 0; i < v.size(); i++) {
        s.push(v[i]);
    }
    for (int i = v.size() - 1; i >= v.size() / 2; i--) {
        s.push(v[i]);
    }
    cout << s << endl;        // (a) Checkpoint 1

    v.add(s.pop());
    v.add(s.pop());
    cout << s << endl;        // (b) Checkpoint 2
    cout << v << endl;        // (c) Checkpoint 3

    Queue<int> q;
    while (!s.isEmpty()) {
        q.enqueue(s.pop());
    }
    cout << q << endl;        // (d) Checkpoint 4

    Vector<int> v2;
    while (!q.isEmpty()) {
        v2.add(q.dequeue());
    }
    cout << v2 << endl;        // (e) Checkpoint 5
}
```

(scratch space–problem 1 continues on next page)

(a) (3pts) Write the output of the line marked Checkpoint 1 in this box:

<i>Bottom</i>	{		}	<i>Top</i>
---------------	---	--	---	------------

(b) (3pts) Write the output of the line marked Checkpoint 2 in this box:

<i>Bottom</i>	{		}	<i>Top</i>
---------------	---	--	---	------------

(c) (3pts) Write the output of the line marked Checkpoint 3 in this box:

{		}
---	--	---

(d) (3pts) Write the output of the line marked Checkpoint 4 in this box:

<i>Front</i>	{		}	<i>Back</i>
--------------	---	--	---	-------------

(e) (3pts) Write the output of the line marked Checkpoint 5 in this box:

{		}
---	--	---

(scratch space—end of problem 1)

(scratch space—exam continues on next page)

2. Big-O (10 points)

Give the Big-O analysis as we learned in class (i.e., tight bound worst-case runtime complexity class) for each of the following code fragments, **in terms of the 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 a precise count*. Write your answer in the boxes on the right side. For each problem, the N that you use for your analysis is defined as the value of the variable N shown in the code. If running the code will make an infinite loop or stack overflow, **write a large X in the answer box**. Each part is worth 2pts.

	Code Fragment	Answer
(a)	<pre>void functionA(Set<int>& s) { int N = s.size(); for (int i = 0; i < N; i++) { cout << s.isEmpty() << endl; } }</pre>	$O(\quad)$
(b)	<pre>void functionB(Set<int>& s) { int N = s.size(); for (int i = 0; i < N; i++) { cout << s.isEmpty() << endl; s.add(i); } }</pre>	$O(\quad)$
(c)	<pre>void functionC(Set<int>& s) { int N = s.size(); Set<int> copy = s; for (int i = 0; i < N; i++) { cout << s.isEmpty() << endl; s.add(i); cout << copy.isSubsetOf(s) << endl; } }</pre>	$O(\quad)$
(d)	<pre>// Assume at start v is nonempty and the values it contains are // some combination of 1's, 3's, and 5's (e.g., {3, 1, 5, 5, 3, 3, 5}) void functionD(Vector<int>& v) { int N = v.size(); for (int i = 0; i < N; i++) { for (int j = 0; j < v[i]; j++) { v.add(0); } } }</pre>	$O(\quad)$
(e)	Analyze the <code>collectionMystery()</code> function in Problem 1 (Stacks and Queues).	$O(\quad)$

(end of problem 2)

3. C++ Strings and Testing Framework: Codebreaking (8 points)

Many historically significant secret code systems relied on letter substitution. For example, an encoding system where each letter is substituted with the one that follows it in the alphabet gives you this:

BOB SAYS BOO → CPC TBZT CPP

To figure out the pattern, codebreakers would often use their knowledge of how often letters and pairs of letters typically occur in English text. For example, letter E is more common than Q, and the substring ST is more common than SF.

To aid in gathering this knowledge, someone has written a function that takes as input a **Vector** of words from the text of thousands of English Wikipedia pages, and is supposed to fill a **Map** with counts of how many times different length 1 and length 2 substrings occur:

```
void makeLetterCounts(Vector<string>& words, Map<string,int>& counts) {
    for (string word : words) {
        for (int i = 0; i < word.length() - 1; i++) {
            counts[word.substr(i,1)]++;
            counts[word.substr(i,2)]++;
        }
    }
}
```

For the original "BOB SAYS BOO" input text above, the function is **intended** to produce the following contents in the Map, for all the length 1 and length 2 substrings in the input (*i.e.*, this is the correct behavior):

counts: {"A":1, "AY":1, "B":3, "BO":2, "O":3, "OB":1,
 "OO":1, "S":2, "SA":1, "Y":1, "YS":1}

But there is a bug in the code! Your job is to use our CS106B C++ testing framework to help demonstrate the code's behavior – both correct and incorrect – to its author.

- (a) (2pts) Start with cheering up the author by showing their code is not *completely* broken. Complete this test case, showing one **length 1** substring for which the expected count is **nonzero** and the code **does** work correctly (*i.e.*, even with the bug, this test should pass).

```
STUDENT_TEST("Length 1 test case that passes.") {
    Map<string, int> counts;
    Vector<string> words = {"BOB", "SAYS", "BOO"};
    makeLetterCounts(words, counts);

    EXPECT_EQUAL(counts[""], );
}
```

(scratch space—problem 3 continues on next page)

- (b) (2pts) Do any length 2 test cases pass? If so, complete this test case, showing one **length 2** substring for which the expected count is **nonzero** and the code **does** work correctly (*i.e.*, even with the bug, this test passes). If not, write “N/A” in both boxes.

```
STUDENT_TEST("Length 2 test case that passes?") {  
    Map<string, int> counts;  
    Vector<string> words = {"BOB", "SAYS", "BOO"};  
    makeLetterCounts(words, counts);  
  
    EXPECT_EQUAL(counts["  "],  );  
}
```

- (c) (2pts) Now **show the author a bug**. Complete this test case, showing one **length 1** substring for which the expected count is **nonzero** and the code **does not** work correctly (*i.e.*, for a working code, this test would pass, but due to the bug, this test will fail).

```
STUDENT_TEST("Length 1 test case that fails!") {  
    Map<string, int> counts;  
    Vector<string> words = {"BOB", "SAYS", "BOO"};  
    makeLetterCounts(words, counts);  
  
    EXPECT_EQUAL(counts["  "],  );  
}
```

- (d) (2pts) Do any length 2 test cases fail? If so, complete this test case, showing one **length 2** substring for which the expected count is **nonzero** and the code **does not** work correctly (*i.e.*, for a working code, this test would pass, but due to the bug, this test will fail). If not, write “N/A” in both boxes.

```
STUDENT_TEST("Length 2 test case that fails?") {  
    Map<string, int> counts;  
    Vector<string> words = {"BOB", "SAYS", "BOO"};  
    makeLetterCounts(words, counts);  
  
    EXPECT_EQUAL(counts["  "],  );  
}
```

(scratch space—end of problem 3)

4. ADTs: LLMs (14 points)

LLMs like Claude and ChatGPT rely on a similar set of statistics as Problem 3's codebreakers to generate text that sounds like typical English. Instead of tracking frequency of pairs of *letters*, they track frequency of pairs of *tokens* (each token is basically a word, and we will only track pairs of tokens, not longer context windows). This problem tests a variety of ADT configurations, including ADTs that hold other ADTs and ADTs that hold structs.

- (a) (7pts) Write a function that takes as input a **Vector** of tokens from the text of thousands of English Wikipedia pages, and fills a **Map** called **counts** recording how many times pairs of consecutive tokens occur.

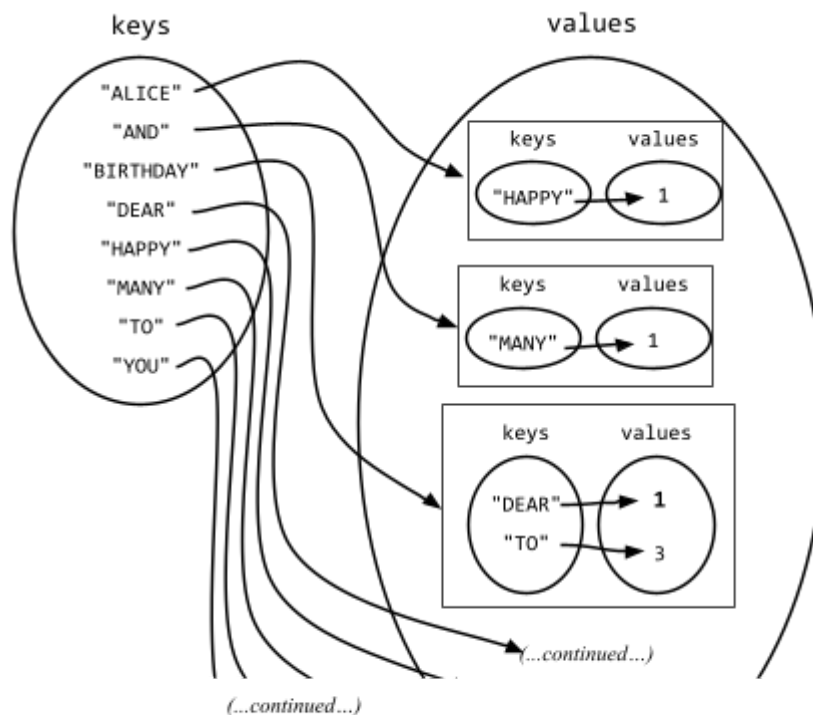
```
void makeWordCounts(Vector<string>& tokens, Map<string, Map<string, int>>& counts);
```

Assume that the tokens are stored in the **Vector** in the same order they appeared in the text (for simplicity, do not worry about how that carries from one Wikipedia page to another, just treat all tokens as consecutive). Assume **counts** is initially empty and you are to fill it, structured as follows:

- Each **key** of **counts** is the **first token in a pair** of consecutive tokens (so the last token of the input will not be counted).
- Each **value** of **counts** is **another Map** where:
 - Each **key** of this map-within-a-map is the **second token in a pair**.
 - Each **value** is a count of **how many times** that token pair was found.
- **Example:** for the text {"HAPPY", "BIRTHDAY", "TO", "YOU", "HAPPY", "BIRTHDAY", "TO", "YOU", "HAPPY", "BIRTHDAY", "DEAR", "ALICE", "HAPPY", "BIRTHDAY", "TO", "YOU", "AND", "MANY", "MORE"}, the function should produce the following:

```
counts = {"ALICE":{"HAPPY":1}, "AND":{"MANY":1}, "BIRTHDAY":{"DEAR":1, "TO":3},  
"DEAR":{"ALICE":1}, "HAPPY":{"BIRTHDAY":4}, "MANY":{"MORE":1}, "TO":{"YOU":3},  
"YOU":{"AND":1, "HAPPY":2}}
```

- **Example:** Here is a **partial diagram** of our example counts, to help you visualize its structure:



(problem 4 continues on next page)

}

(scratch space—problem 4 continues on next page)

(b) (7pts) Write a function that takes as input the counts from part (a), and further refines its data for use as an LLM.

```
void makeLLM(Map<string, Map<string, int>>& counts, Map<string, Vector<Next>>& LLM);
```

Specifically, it should go through all the keys of `counts` and, for each key, calculate the probability of each of the tokens that might follow it. Consider the output from the example in part (a) now as input:

```
counts = {"ALICE":{"HAPPY":1}, "AND":{"MANY":1}, "BIRTHDAY":{"DEAR":1, "TO":3},
"DEAR":{"ALICE":1}, "HAPPY":{"BIRTHDAY":4}, "MANY":{"MORE":1}, "TO":{"YOU":3},
"YOU":{"AND":1, "HAPPY":2}}
```

The probabilities of the tokens that follow "BIRTHDAY" are: "DEAR", 0.25 and "TO", 0.75, because $1 / \text{double}(1 + 3) = 0.25$ and $3 / \text{double}(1 + 3) = 0.75$. You will fill a Map called LLM with these calculated probabilities. Like with `counts`, each key will be a token, but the values will be `Vectors` instead of more Maps.

- You may assume that `counts` is initially filled as described in part (a) and LLM is initially empty.
- Recall that dividing one `int` by another `int` will produce an `int`, so you must ensure `double`.
- Each **key** of LLM is the **first token in a pair** of consecutive tokens (same keys as `counts`).
- Each **value** of LLM is a `Vector<Next>`, where each item in the `Vector` is a `struct Next` that holds one of the tokens that follows the key and the probability you calculate for it:

```
struct Next {
    string token; // As usual for a struct, you may use a constructor
    double prob;  // and/or access fields directly using . operator
    Next(string t, double p);
};
```

- So, for the input shown above, the function should produce the following:

```
LLM = {"ALICE":{"("HAPPY",1)}, "AND":{"("MANY",1)}, "BIRTHDAY":{"("DEAR",0.25),
("TO",0.75)}, "DEAR":{"("ALICE",1)}, "HAPPY":{"("BIRTHDAY",1)}, "MANY":{"("MORE",1)},
"TO":{"("YOU",1)}, "YOU":{"("AND",0.333333), ("HAPPY",0.666667)}}
```

(scratch space—write your solution on the next page)

(end of problem 4)

(scratch space—end of exam)