# Practice Midterm 5

CS106B, Spring 2024

_____

(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. C++ and ADTs (30 pts)**

Write a function that takes a vector of strings – where each string contains a single sentence – and returns a map of type *Map<string, Map<string, int>>* where *map*[*s1*][*s2*] is the number of sentences in the vector that contain **both** *s1* and *s2*.

Let's unpack that problem statement together and explore some particulars:

First of all, each string in the vector will contain a single sentence where all characters have been converted to lowercase and all punctuation has been removed. Within each string, words are separated with a single space. For example, the following vector contains three strings (three sentences):

   { "i like tea",  "i like cupcakes",  "i like cherry tea and cherry cupcakes" }

In the above vector, there is exactly one sentence that contains both "cupcakes" and "tea," and so our map would have:

   map["cupcakes"]["tea"] = map["tea"]["cupcakes"] = 1

Similarly, there are three sentences that contain both "i" and "like," and so our map would have:

   map["i"]["like"] = map["like"]["i"] = 3

Take special note of the fact that there is exactly one sentence that contains both "like" and "cherry." Even though that sentence contains two instances of the word "cherry," we only count that sentence once, and so our map would have:

   map["like"]["cherry"] = map["cherry"]["like"] = 1

You might be wondering about that *Map<string, Map<string, int>>* type. Here's how that works: given a key, *s1*, this map unlocks a submap. If we feed the key *s2* to that submap, we should get an integer indicating the number of sentences in which both *s1* and *s2* appear. We access that particular integer using *map*[*s1*][*s2*].

For example, the full map for the vector of sentences above should look like so:

```
{
    "and"   :  { "cherry" : 1,  "cupcakes" : 1,  "i" : 1,  "like" : 1,  "tea" : 1 },
    "cherry"   :  { "and" : 1,  "cupcakes" : 1,  "i" : 1,  "like" : 1,  "tea" : 1 },
    "cupcakes"   :  { "and" : 1,  "cherry" : 1,  "i" : 2,  "like" : 2,  "tea" : 1 },
    "i"   :  { "and" : 1,  "cherry" : 1,  "cupcakes" : 2,  "like" : 3,  "tea" : 2 },
    "like"   :  { "and" : 1,  "cherry" : 1,  "cupcakes" : 2,  "i" : 3,  "tea" : 2 },
    "tea"   :  { "and" : 1,  "cherry" : 1,  "cupcakes" : 1,  "i" : 2,  "like" : 2 }
}
```

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

- Your function must be **iterative** (not recursive) to earn credit.
- You cannot create any helper functions. All your work must be done in a **single function**.
- You must use the function signature given below.
- Your function should not alter the contents of the vector in any way. When the function returns, the vector should be just as it was when passed to the function.
- Note that the map should be symmetric: *map*[*s1*][*s2*] should always equal *map*[*s2*][*s1*].

- We should never place a string in its own submap. For example, *map*["pie"]["pie"] = 0 (not 1).
- Trust the *map*[*s1*][*s2*] syntax. If you use it, it will probably work exactly as you expect it to.
- Recall that *map*[*key*] automatically adds the key to the map – no need for manual initialization. It Just Works™.
- For full credit, you must think of an efficient way to ensure we don't double-increment a value when a sentence contains duplicate strings. For example, in the third sentence above, when we hit the second occurrence of "cherry," it would be inefficient to loop through the previous words one-by-one to see if we had encountered "cherry" earlier in the sentence.

```
Map<string, Map<string, int>> coOccurrenceMap(Vector<string>& sentences) {
```

**2. Big-O (30 pts)** Consider the following functions:

```cpp
// Removes the top-most instance of the given target integer from the stack (if
// present). Returns true if the target is found, false otherwise.
bool seekAndDestroy(Stack<int>& original, int target)
{
    bool found = false;
    Stack<int> sideDish;

    while (!original.isEmpty())
    {
        int temp = original.pop();
        if (temp == target)
        {
            cout << "Target found!" << endl;
            found = true;
            break;
        }
        sideDish.push(temp);
    }

    while (!sideDish.isEmpty())
    {
        original.push(sideDish.pop());
    }

    return found;
}

void removeAllInstances(Stack<int>& s, int target)
{
    // Calls function repeatedly, removing all instances of target.
    while (seekAndDestroy(s, target))
    {
        // Does nothing here. Just loop back around until function returns false.
    }
}
```

a) Suppose we call *seekAndDestroy()* with the following input stack and **target = 80**:
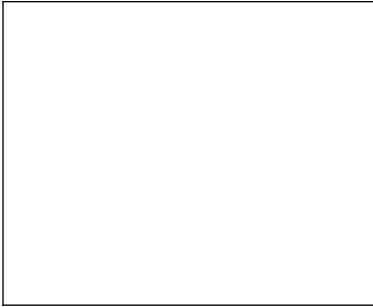
| **Original Stack** (*original*) | Draw a vertical stack (like the one to the left) showing the contents of *original* at the moment when "Target found!" gets printed: | Draw a vertical stack (like the one to the left) showing the contents of *sideDish* at the moment when "Target found!" gets printed: |
|---|---|---|
| ```
+----+
| 33 | <- top
+----+
| 61 |
+----+
| 80 |
+----+
| 92 |
+----+
| 17 |
+----+
| 48 |
+----+
| 53 | <- bottom
+----+
``` | | |

**Note:** The next four questions focus on the **first** function above: *seekAndDestroy()*.  **Please place answers in boxes!**

b)  What is the **worst-case** Big-O runtime for *seekAndDestroy()*, assuming it receives a stack with *n* elements?

O(            )

c)  Draw a stack with 5 elements that causes *seekAndDestroy()* to encounter its **worst-case** runtime when *target* **= 15**.

d)  What is the **best-case** Big-O runtime for *seekAndDestroy()*, assuming it receives a stack with *n* elements?

O(            )

e)  Draw a stack with 5 elements that causes *seekAndDestroy()* to encounter its **best-case** runtime when *target* **= 15**.

**Note:** These questions focus on the **second** function above: *removeAllInstances()*.  **Please place answers in boxes!**

f)  What is the runtime for *removeAllInstances()* if it receives a stack where **all *n* of its values** are equal to the *target*?

O(            )

g)  What is the runtime for *removeAllInstances()* if it receives a stack where **none of its *n* values** are equal to the *target*?

O(            )

h)  What is the runtime for *removeAllInstances()* if it receives a stack with *n* values, where the top *k* values are **not** equal to the target, and the bottom *m* values **are** equal to the target (and *n = k + m*)?

O(            )

**3. Recursion (20 pts)**

Write a recursive function that takes a single non-negative integer, *n*, and returns the sum of all of its digits. For example:

```
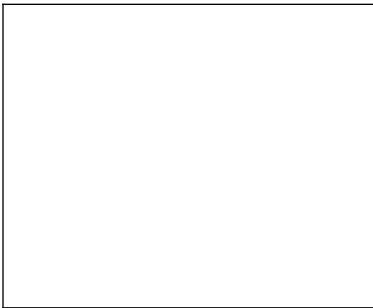digitSum(12538)  // Returns 1 + 2 + 5 + 3 + 8 = 19
digitSum(3)      // Returns 3
digitSum(0)      // Returns 0
```

You may assume we will never pass a negative integer to your function.

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 call any functions from the *math.h* library (such as an exponentiation function).
5. You cannot create any strings. If you find yourself using strings or ASCII values, you are on the wrong track.
6. You cannot modify the function signature given below.
7. Each call to your recursive function must process only a **single digit** from the given integer and rely on subsequent recursive calls to handle the rest.

**Hint:** To get the ones digit of a number, mod by 10. To remove the ones digit, divide by 10.

The function signature is:

```
int digitSum(int n) {
```

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

*We will not grade anything on this page.*

## 4. Recursive Backtracking (20 pts)

In this problem, you will be given a vector of *fruitT* structs.[1] The struct definition is as follows:

```
struct fruitT
{
    string label;  // Name of fruit.
    int count;     // Number available.
};
```

Write a function that takes the following parameters and returns the number of fruit baskets we can create that satisfy those parameters:

> *Vector<fruitT>& fruits* : The vector of fruits available for constructing the fruit basket.
> *int minFruit* : The minimum number of fruits we must add to our basket (otherwise, it looks too empty).
> *int maxFruit* : The maximum number of fruits we can add to our basket (otherwise, it gets too full).
> *int minTypes* : The min number of different *types* of fruit we need in our basket (otherwise, the basket is too plain).

Note that we are interested simply in the number of each kind of fruit and not the order in which they are chosen. So, selecting two apples followed by one pear is the same as selecting one pear followed by two apples. Similarly, all fruits of the same type should be considered equal. So, if we select two apples for our basket, it doesn't matter *which* two apples we take – only that there are two apples total.

For example, suppose we make the following call to our function:

> *countBaskets*( {{"kiwi", 2}, {"pear", 3}}, *minFruit* = 3, *maxFruit* = 4, *minTypes* = 1 )

There are 5 possible baskets we can create:

> {2 kiwis + 1 pear},  {1 kiwi + 2 pears},  {3 pears},  {2 kiwis + 2 pears},  {1 kiwi + 3 pears}

Similarly, suppose we make the following call to our function:

> *countBaskets*( {{"kiwi", 2}, {"pear", 3}}, *minFruit* = 3, *maxFruit* = 4, **minTypes = 2** )

There are now only 4 possible baskets we can create, because every basket must have at least two types of fruit:

> {2 kiwis + 1 pear},  {1 kiwi + 2 pears},  {2 kiwis + 2 pears},  {1 kiwi + 3 pears}

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

- Your algorithm must be **recursive** and must use **backtracking** techniques to generate its results.
- You must not create any auxiliary data structures! (No new vectors.)
- You can never modify the contents of a *fruitT* struct. You may add and remove structs from the *fruits* vector, but you cannot change the contents of an individual struct. You also cannot create copies of any structs.

---

1   Probably the most fun thing about the *fruitT* struct is that it can be pronounced "fruit tea" or "fruity." Either one works! Enjoy!

- You **must** remove elements from the *fruits* vector as you go. Do not use the approach we have seen in class where we use an independent variable, *k*, to move through the vector.

- Notice that the vector is passed by reference, so if you remove elements from the vector, you must add them back later. When your function terminates, the vector should be back in the original state it was in when the function was called. (The elements should not be modified, and they should not be in a different order.)

- 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 can use a one-line wrapper function that leads into a recursive helper function if you wish, but beyond that, you cannot write any other functions. (There is a way to solve this problem without a wrapper, as well!)

For your reference, here is a *countSubets()* function that uses backtracking to count the number of subsets we can generate that have at least *minSize* elements from the given vector. Read over this code as a starting point. You can borrow code and structure from *countSubsets()* when writing *countBaskets()*, but be warned that there is a key inefficiency with this approach that you should avoid in your *countBaskets()* solution, and it also ruins the order of the elements in the vector.

```cpp
int countSubsets(Vector<int>& v, int minSize, int sizeSoFar) {
    if (v.size() == 0) {
        if (sizeSoFar >= minSize) {
            return 1;
        } else {
            return 0;
        }
    }

    int total = 0;
    int thisOne = v.remove(0);

    total += countSubsets(v, minSize, sizeSoFar + 1);  // include thisOne in subset
    total += countSubsets(v, minSize, sizeSoFar);      // don't include thisOne

    v.add(thisOne);
    return total;
}

// wrapper function
int countSubsets(Vector<int>& v, int minSize)
{
    return countSubsets(v, minSize, 0);
}
```

*The function signature is given on the following page.*

```cpp
int countBaskets(Vector<fruitT>& fruits, int minFruit, int maxFruit, int minTypes) {
```