Elyse Cornwall
Amrita Kaur

July 17, 2023

CS106B Midterm Exam

You have 2 hours (120 minutes) to complete this exam. The exam is 120 points, meant to correspond to one point per minute of the exam. You may want to use the point allocation for each problem as an indicator for pacing yourself on the exam.

This is a closed-book, closed-device exam. The only materials you may consult during this exam are the provided reference sheet and one 8.5" by 11" inch page (front and back) of notes.

The majority of the points on this exam will be awarded for showing a mastery of the concepts taught in CS106B. There is no guarantee that you will receive points for using concepts taught in prior classes. You do not need to `#include` any libraries, comment your code, or worry about minor syntactical issues. You may use helper functions if they help you solve the problem, but decomposition is not required and is not necessary to solve any of these problems.

We will only grade one answer per problem. We cannot grade multiple solution attempts for a single problem, so please clearly indicate your final solution if there are any notes or previous attempts on the page.

If you need extra space to work on a problem, you may use scratch paper. However, we will not grade or accept any work done on scratch paper. All final answers must be written on the exam booklet in the designated sections.

Problem	**Points**
1. C++ Fundamentals	20
2. C++ Tracing	20
3. Collections	30
4. Big-O and Algorithmic Analysis	20
5. Recursive Tracing	10
6. Recursion	20
Total	120

I acknowledge and accept the letter and spirit of the honor code.

Signature: _____

Name (print): _____

SUNetID (i.e. your @stanford.edu email): _____

1 C++ Fundamentals [20 points]

Implement the function:

```
bool verifySentence(string sentence)
```

that checks whether a string representing a sentence is valid. Here are the rules for a valid sentence:

1. The first character of the sentence must be capitalized.
2. The last character of the sentence must be an end punctuation mark (either `'.'`, `'!'`, or `'?'`).
3. Any other characters between the first and last must be lowercase, or a space.

Here are a few example calls to the `makeSentence` function.

- `verifySentence("You got this!")` returns `true`
- `verifySentence("You got this")` returns `false` (no end punctuation)
- `verifySentence("You Got This!")` returns `false` (incorrect sentence capitalization)
- `verifySentence("you got this!")` returns `false` (incorrect sentence capitalization)
- `verifySentence("You!")` returns `true`

You can assume the following:

- `sentence` will only contain alphabetical characters, spaces, and end punctuation. The end punctuation, if any, will always occur as the last character of the string.
- `sentence` will be at least 2 characters long.
- You can use any of the C++ and/or Stanford library functions to solve this problem, and you don't need to write `#include` statements for these. *Hint: the `isupper` and `islower` functions may be helpful, but note that they return `false` for any non-alphabetical characters.*

You can use the rest of this page for scratch work, which will not be graded. Please write your solution on the following page.

Please write your solution to Problem 1 on this page.

```
bool verifySentence(string sentence) {
    for (int i = 0; i < sentence.length(); i++) {
        char letter = sentence[i];
        if (i == 0) {
            if (!isupper(letter)) {
                return false;
            }
        } else if (i > 0 && i < sentence.length() - 1) {
            if (letter != ' ' && !islower(letter)) {
                return false;
            }
        }
        else if (letter != '.' && letter != '!' && letter != '?') {
                return false;
        }
    }
    return true;
}
```

```
// ALTERNATE SOLUTION

bool verifySentence(string sentence) {
    char firstLetter = sentence[0];
    if (!isupper(firstLetter)) {
        return false;
    }

    char lastLetter = sentence[sentence.length() - 1];
    if (lastLetter != '.' && lastLetter != '!' && lastLetter != '?') {
        return false;
    }

    for (int i = 1; i < sentence.length() - 1; i++) {
        char letter = sentence[i];
        if (letter != ' ' && !islower(letter)) {
            return false;
        }
    }

    return true;
}
```

2 C++ Tracing [20 points]

In this problem, we're going to explore the following *correctly implemented* function:

```
void cumulative(Vector<int>& vec) {
    for (int i = 1; i < vec.size(); i++) {
        vec[i] = vec[i] + vec[i - 1];
    }
}
```

This function takes in a reference to a Vector `vec` and modifies `vec` so that each index contains the cumulative sum of integers thus far. Note that this function doesn't return anything; instead, it modifies the Vector `vec`. Although the code above would correctly handle negative values, we'll assume that the integers in `vec` are always non-negative.

Here are a few example calls to show how the `cumulative` function modifies different input vectors:

Original value of `vec`	Value of `vec` after calling `cumulative(vec)`
`{1, 1, 1, 1, 1}`	`{1, 2, 3, 4, 5}`
`{1, 2, 3, 4, 5}`	`{1, 3, 6, 10, 15}`
`{0, 5, 0, 0}`	`{0, 5, 5, 5}`

Now, we will explore a few variations of `cumulative` and how they behave.

Part A

Here's a variation of `cumulative` where the ampersand in the function definition is removed.

```
void cumulativeA(Vector<int> vec) { // ampersand removed from this line
    for (int i = 1; i < vec.size(); i++) {
        vec[i] = vec[i] + vec[i - 1];
    }
}
```

The intended functionality is for `cumulativeA` to behave just like `cumulative` above. Does this function produce the same result given the input `vec = {1, 2, 3, 4, 5}`? If not, what is the value of `vec` after calling `cumulativeA(vec)` with this example Vector?

Answer YES/NO for Part A, and give value of `vec` if NO:

NO
{1, 2, 3, 4, 5}

Part B

Here's a variation of `cumulative` where the for loop is changed: the variable `i` is initialized to 0 instead of 1, and the condition is `i < vec.size() - 1` instead of `i < vec.size()`. Additionally, the code inside of the for loop has been changed from `vec[i - 1]` to `vec[i + 1]`.

```
void cumulativeB(Vector<int>& vec) {
    for (int i = 0; i < vec.size() - 1; i++) { // for loop changed
        vec[i] = vec[i] + vec[i + 1]; // + 1 instead of -1
    }
}
```

The intended functionality is for `cumulativeB` to behave just like `cumulative` on the previous page. Does this function produce the same result given the input `vec = {1, 2, 3, 4, 5}`? If not, what is the value of `vec` after calling `cumulativeB(vec)` with this example Vector?

Answer YES/NO for Part B, and give value of vec if NO:

NO
{3, 5, 7, 9, 5}

Part C

Here's a variation of `cumulative` where the input is represented as a Queue rather than a Vector.

```
void cumulativeC(Queue<int>& q) {
    int queueSize = q.size();
    int sum = q.dequeue();
    q.enqueue(sum);
    for (int i = 1; i < queueSize; i++) {
        int cur = q.dequeue();
        sum += cur;
        q.enqueue(sum);
    }
}
```

The intended functionality is for `cumulativeC` to behave much like `cumulative` on page 4, but rather than modifying a Vector `vec` to store the running sum from index 0 to the end of the Vector, we'd like to modify a Queue `q` to store the running sum from the front to the back of the Queue. Does this function produce the desired result given the input `q = {1, 2, 3, 4, 5}` (1 being the front, 5 being the back of the Queue)? If not, what is the value of `q` after calling `cumulativeC(q)` with this example Queue?

Answer YES/NO for Part C, and give value of `q` if NO:

YES

Part D

Here's a variation of `cumulative` where the input is represented as a Stack rather than a Vector.

```
void cumulativeD(Stack<int>& s) {
    int stackSize = s.size();
    int sum = s.pop();
    s.push(sum);
    for (int i = 1; i < stackSize; i++) {
        int cur = s.pop();
        sum += cur;
        s.push(sum);
    }
}
```

The intended functionality is for `cumulativeD` to behave much like `cumulative` on page 4, but rather that modifying a Vector `vec` to store the running sum from index 0 to the end of the Vector, we'd like to modify a Stack `s` to store the running sum from the top to the bottom of the Stack. Does this function produce the desired result given the input `s = {5, 4, 3, 2, 1}` (1 being the top, 5 being the bottom of the Stack)? If not, what is the value of `s` after calling `cumulativeD(s)` with this example Stack?

Answer YES/NO for Part D, and give value of `s` if NO:

NO
{5, 4, 3, 2, 16}

3 Collections [30 points]

As a budding foodie, you've made it your mission this summer to explore all of the best doughnut places around the Bay. You notice that many of the shops sell the same flavors, but occasionally, they'll make a speciality doughnut flavor that you've never had before!

Part A

Implement the function:

```
Map<string, Set<string>> flavors(Map<string, string>& menus)
```

that takes in one parameter: a reference to a Map where each key is the name of a doughnut shop (represented as a string) and each value is a series of flavors sold at that shop (represented as a comma-separated string). Your function should return a Map where each key is a doughnut flavor from `menus` (as a string), and each value is all the shops that sell that flavor (as a Set of strings).

For example, let's say the Map `menus` stores the following information:

```
{
     "Daily Donuts": "apple cinnamon, chocolate, glazed",
     "Happy Donuts": "apple cinnamon, glazed, maple, vanilla",
     "Stan's": "apple cinnamon, chocolate, glazed, vanilla"
}
```

The call to the function `flavors(menus)` should return the Map:

```
{
     "apple cinnamon": {"Daily Donuts", "Happy Donuts", "Stan's"},
     "chocolate": {"Daily Donuts", "Stan's"},
     "glazed": {"Daily Donuts", "Happy Donuts", "Stan's"},
     "maple": {"Happy Donuts"},
     "vanilla": {"Happy Donuts", "Stan's"}
}
```

Some notes on this problem:

- You may assume that every shop in `menus` is selling at least one flavor.
- Notice that the flavors in the values of `menus` are separated by both a comma and a space.
- You may assume that no flavor name has a comma within it.
- You can use any of the C++ and/or Stanford library functions to solve this problem, and you don't need to write `#include` statements for these. *Hint: the `stringSplit` function may be useful here.*

You can use this page for scratch work, which will not be graded. Please write your solution on the following page.

Please write your solution to Problem 3, Part A on this page.

```
Map<string, Set<string>> flavors(Map<string, string>& menus) {
    Map<string, Set<string>> result;
    for (string store: menus) {
        string value = menus[store];
        Vector<string> donuts = stringSplit(value, ", ");
        for (string donut: donuts) {
            result[donut].add(store);
        }
    }
    return result;
}
```

}

Part B

This problem builds on the Map returned from `flavors` in Part A, but both parts may be completed individually. You do not need a working implementation for Part A to get full points for Part B.

Implement the function:

```
string mostUnique(Map<string, Set<string>>& flavors)
```

that takes in one parameter: a Map where each key is a doughnut flavor (as a string) and each value is all the doughnut shops that sell that flavor (as a Set of strings). Your function should return the name of the most unique flavor in the Map, which is the doughnut flavor that is sold at the fewest number of shops.

For example, let's say the Map `flavors` stores the following information:

```
{
     "apple cinnamon": {"Daily Donuts", "Happy Donuts", "Stan's"},
     "chocolate": {"Daily Donuts", "Stan's"},
     "glazed": {"Daily Donuts", "Happy Donuts", "Stan's"},
     "maple": {"Happy Donuts"},
     "vanilla": {"Happy Donuts", "Stan's"}
}
```

The call to `mostUnique(flavors)` should return `"maple"`. This is because maple doughnuts are only sold at one shop, while all the other flavors are sold at at least two shops.

Some notes on this problem:

- If multiple flavors are tied for most unique, you may return any one of these flavors.
- You may assume that `flavors` is a non-empty Map and that every doughnut flavor in the Map is sold at at least one shop.
- You can use any of the C++ and/or Stanford library functions to solve this problem, and you don't need to write `#include` statements for these.

You can use this page for scratch work, which will not be graded. Please write your solution on the following page.

Please write your solution to Problem 3, Part B on this page.

```
string mostUnique(Map<string, Set<string>>& flavors) {
    int best = -1;
    string mostUniqueFlavor;
    for (string flavor: flavors) {
        int count = flavors[flavor].size();
        if (count < best || best == -1) {
            best = count;
            mostUniqueFlavor = flavor;
        }
    }
    return mostUniqueFlavor;
}
```

}

4 Algorithmic Analysis and Big-O [20 points]

Give the Big-O time complexity for each of the following code snippets, in terms of the variable `N`. Remember that Big-O describes how the run time of the code snippet grows with the size of the input `N`. You must provide a simplified expression in Big-O notation, removing any lower order terms and leading coefficients (e.g. `O(`N^3`)` rather than `O(`$2N^3 + 4N + 13$`)`).

For each problem, you may assume that you have the following Stanford Library data structures, each with `N` elements in them (except the Grid, which will have $N \times N = N^2$ elements). These data structures have been created and filled with `N` elements for you. Each of the subproblems below is **independent**, meaning that changes that occur to these structures in one subproblem do not carry over into other subproblems.

```
Vector<int> myVector;
Queue<int> myQueue;
Stack<int> myStack;
Set<int> mySet;
Grid<int> myGrid;
```

Part A

```
for (int elem: myGrid) {
    cout << elem + 10 << endl;
}
```

Part A answer:

O(N^2)

Part B

```
while (!myStack.isEmpty()) {
    int elem = myStack.pop();
    if (elem % 2 == 0) {
        cout << "even" << endl;
    } else {
        cout << "odd" << endl;
    }
}
```

Part B answer:

O(N)

Part C

```
for (int i = 0; i < 3; i++) {
    for (int j = 0; j < myQueue.size(); j++) {
        int elem = myQueue.dequeue();
        myQueue.enqueue(elem);
    }
}
```

Part C answer:

O(N)

Part D

```
for (int elem: myVector) {
    if (mySet.contains(elem)) {
        cout << "found it!" << endl;
    }
}
```

Part D answer:

O(N LOG(N))

5 Recursive Tracing [10 points]

In this problem, you will trace through two recursive "mystery functions." Read through each function and answer the questions below.

Part A

```
int recursionTraceA(int n) {
    if (n == 1) {
        return 0;
    } else {
        return 1 + recursionTraceA(n / 2);
    }
}
```

Note: you can assume that `n` will always be greater than or equal to 1.

What is the return value of `recursionTraceA(16)`?

Answer:

4

What mathematical operation does `recursionTraceA(n)` compute for some integer `n`?

Answer:

LOG(N)

Part B

```
bool recursionTraceB(string s) {
    if (s.length() <= 1) {
        return true;
    } else {
        if (s[0] <= s[1]) {
            return recursionTraceB(s.substr(1, s.length() - 1));
        } else {
            return false;
        }
    }
}
```

Note: the string `s` will only contain lowercase alphabetical characters. As a reminder, characters in C++ have an underlying ASCII numerical representation. The following is an abridged version of an ASCII chart that maps characters to their numerical representations.

Char	Decimal	Char	Decimal	Char	Decimal	Char	Decimal
a	97	h	104	o	111	v	118
b	98	i	105	p	112	w	119
c	99	j	106	q	113	x	120
d	100	k	107	r	114	y	121
e	101	l	108	s	115	z	122
f	102	m	109	t	116		
g	103	n	110	u	117		

What is the return value of `recursionTraceB("almost")`?

Answer:

`true`

In your own words, what is `recursionTraceB` doing?

Answer:

Checking whether s is in alphabetical order.

6 Recursion [20 points]

As a lover of both literature and computer science, you decide to implement the following function:

```
bool doesMatch(Vector<string>& words, Queue<int>& numbers)
```

This function takes in two parameters: a Vector of strings populated with single words, and a Queue of integers populated with numbers. Your function will check whether the number of characters in each element of `words` equals the value of the next element in `numbers`. If this holds, your function will return `true`. Otherwise, it will return `false`.

Here are some additional stipulations:

- **You must use recursion** to solve this problem. No looping allowed.
- If an element in `words` is an empty string, this counts as a string of length 0 (though you shouldn't need to handle this case separately).

For example, let's say we had the following variables:

```
words1 = {"For", "a", "time", "I", "lived", "radiantly", ""};
numbers1 = {3, 1, 4, 1, 5, 9, 0};
```

The call to the function `doesMatch(words1, numbers1)` should return `true`.

As another example, let's say we had the following variables:

```
words2 = {"You", "are", "almost", "there!"};
numbers2 = {3, 3, 4, 6};
```

The call to the function `doesMatch(words2, numbers2)` should return `false`, since the length of `"almost"` is 6, not 4.

You may assume the following:

- `words` and `numbers` have the same length (there are as many strings in `words` as there are integers in `numbers`).
- All integers in `numbers` are non-negative.
- You may modify `words` and `numbers` however you like, and you don't need to restore them to their original state.
- You can use any of the C++ and/or Stanford library functions to solve this problem, and you don't need to write `#include` statements for these.

You can use this page for scratch work, which will not be graded. Please write your solution on the following page.

Fun fact: Fans of constrained writing, a literary technique in which the writing must adhere to a specific pattern, may have heard of the book *Not a Wake* by Michael Keith. In Keith's 10,000-word book, the number of letters in every word corresponds to the digits of π in order!

Please write your solution to Problem 6 on this page.

```
bool doesMatch(Vector<string>& words, Queue<int>& numbers) {
    if (words.isEmpty() && numbers.isEmpty()) {
        return true;
    }
    string word = words.remove(0);
    int num = numbers.dequeue();
    if (word.length() == num) {
        return doesMatch(words, numbers);
    } else {
        return false;
    }
}
```

}