Final Report

Name: (Tripp) Milton Lamb

A#: 25002371

I wrote the c++ code in vscode on windows and then moved it over to Linux to double check compilation. Typora was used to write markdown and generate pdfs of this report. I am more familiar with Python, JS, Fortran, and Rust so Claude AI was used for syntax lookups and idiomatic c++ patterns. I use windsurf for autocomplete. All logic is my own.

Problem 1

Algorithm Description

I was tasked with writing an algorithm for taking in a sorted list of distinct or indistinct integers and returning the first magic index (or the first index that whose value matches the current index) in sorted order. The program should return "None" if there was no magic index found. Since there is no information given in the input as to the distinctness of the data to follow, I opted to write a simple algorithm that can handle both distinct and indistinct integer input arrays.

The algorithm simply loops through the list from index 0 on checking if the current value of the array equals the current index of the array. I exits when found, guaranteeing to return the first magic index of a sorted array regardless of integer distinctness. It also offers an early exit if the current value exceeds the length of the array. Since the array is sorted if the value exceeds the length (or meets in this case for 0-indexed), even if the rest of the array is duplicate integers a magic index can still never form.

The time complexity is `O(n)` for best, worst, and average case. The application wrapping the algorithm converts a `-1` return value into a 'None' as output to support the requirements document.

Execution Instructions

Build:

```
g++ final_problem_1.cpp -o final_problem_1
```

Execute:

```
./final_problem_1 <input_file_name>
```

Execution Results

Example input:

```
-10 -5 0 3 7
0 2 3 4 5
-1 0 3 5 6
-1 0 2 2 4 6
```

The execution results from standard out for the example input from the assignment are:

```
3
0
None
2
```

Special Notes

If distinctness was given by the input you could reduce the time complexity of the distinct case to `O(log(n))`. You could make a check for distinctness @ `O(n)` time complexity, but then even though the magic index algorithm is `O(log(n))` to total would still be `O(n)`.

This program uses `std::list` internally to read in the data, so the code should allow for any size of array up to computer ram (or process ram) allocation. It will also fail if the input integer is outside the 32-bit integer range (-2,147,483,648 to 2,147,483,647). It could easily be modified to support `long`. The code's implementation of the algorithm described above takes in a list as the input.

Empty lines are treated as empty arrays so you will get 'None' back for empty lines. There should be no logic based limit to the number of lines allowed in an input file.

Problem 2

Algorithm Description

I was tasked with merging two sorted arrays into a single sorted array. The merging was to be done in place on the first array which is initialized to be the size of both arrays combined. The extra space on the first array is not initialized in any particular way.

Given two sorted 0-indexed arrays where the first array (`A`) has an additional trailing buffer the size of the second array (`B`) the algorithm successfully merges the two arrays in sorting order without introducing dynamic memory requirements on top of the memory required to store the original arrays (plus buffer). To avoid having to shift the array values we start at the end array `A`'s buffer, and fill array `A` in reverse order. The largest non-appended values from both array `A` and array `B` and constantly compared and placed in the previous index from the last placed. If one array is depleted, there is additional logic to catch that and shift the rest of the other array to it's correct place.

Execution Instructions

Build:

```
g++ final_problem_2.cpp -o final_problem_2
```

Execute:

```
./final_problem_2 <input_file_name>
```

Execution Results

Example Input file:

```
10 4
3 6 8 15 21 23
2 14 17 25
```

Example standard out:

```
2 3 6 8 14 15 17 21 23 25
```

Special Notes

The algorithm itself offers no validation or safety. The algorithm assumes absolutely correct input. The wrapping program offers all validation. The wrapping program offers all validation for the input values.

Assuming the format is correct the program should not unexpectedly fail. The code will give an erroneous error message if there is an incorrect number of values on the first line. This could be fixed by reading in the first line as a string an performing additional validation.

The code will give useful error messages if

- if the command line argument for file name is not present
- the desired file is not present
- the first value of the first line does not equal the total number of integers in lines 2 and 3
- the number of integers in the first line is greater than the first value of the first line
- the second value of the first line does not equal the number of integers in the third line

Appendix

I could not tell if the algorithm description was supposed to contain the actual algorithm or not, but I could not fit it into a quarter page if so. I have left the algorithms below.

Algorithm 1

The steps for my Find First Magic Index algorithm are as follows:

1. determine array length and store as `n`
2. loop through array by index `i`

   2.a. set `value` equal to current index of the array

   2.b if `value == i` then return `i`. exit algorithm. else proceed

   2.c if `value >= n` then return `-1`. exit algorithm. else proceed

   2.d. increment to next index and proceed back to 2.a.
3. if loop exits normally return `-1` as no magic index exists

Algorithm 2

1. initialize three indices `i_A` to keep track of array `A`, `i_B` to track array `B` and `i` to keep track of the full array `A` including the buffer.

   1.a. Set `i_A` to the length of array `A` - `1`.

   1.b. Set `i_B` to the length of array `B` - `1` .

   1.c. Set `i` to the length of the full array `A` plus buffer length - `1` or more simply `i = len(A) + len(B) - 1`
2. set a loop to iterate while `i >= 0`

   2.a. if `i_A < 0` then set `A[i] = B[i_B]`, decrement `i_B` by one, increment `i` by `1` then restart loop. Else, proceed to 2.b.

   2.b. if `i_B < 0` then set `A[i] = A[i_A]`, decrement `i_A` by one, increment `i` by `1` then restart loop. Else, proceed to 2.c.

   2.c. If `A[i_A] >= B[i_B]` then set `A[i] = A[i_A]`, decrement `i_A` by `1`, increment `i` by `1`, then restart loop. Else, proceed to 2.d.

   2.d. if `A[i_A] < B[i_B]` then set `A[i] = B[i_B]`, decrement `i_B` by `1`, increment `i` by `1`, then restart loop.