Lab 2 Activities [30 Marks]

An Autograded Evaluation

For the lab portion submitted for autograding, do not use `printf()` or `scanf()`.

IMPORTANT: **Your submission must compile without syntactic errors to receive grades. Non-compilable solutions will not be graded.**

About Lab 2

- Use the following link to accept the lab invitation: https://classroom.github.com/a/Nfu6-lKR
- Lab 2 is due on **Friday, October 11, 2024, at 11:59 pm** on Github.
- **You may be forced to deal with a syntactic bug in Q4a first; the bug is part of Q4a debugging activity. If you want to work on other questions first, you can choose to temporarily comment out the provided Q4a code.**

- You are allowed to use functions from <stdio.h> and <math.h>, both of which are included in the skeleton code.
- **Other C Libraries are NOT PERMITTED.**
  Global Variables are NOT PERMITTED.
  Including them will result in zero for the question.

- **Note: You must provide your student information (Section, Mac ID, and Student ID) in the README file of the Lab 2 code package. Failure to do so will result in a 5% penalty on the entire Lab 2 grade.**

- **Lab 2 comes with launch.json to enable the VSCode debugger.**
  You need to set the workspace to your Lab 2 folder to start the debugging session.

Lab Question 1 – Vector Operations [6 marks]

Consider representing vectors with ***n*** floating point components using arrays. Develop a vector operations library containing the following functions using the provided `Question1.c` skeleton code.

- `void add_vectors(double vector1[], double vector2[], double vector3[], int size)`
  - This function performs the Vector Sum operation on two vectors of the same dimension.
  - `vector3` should store the result of vector sum of `vector1` and `vector2`.
  - You can assume that all three arrays have the same size defined by input parameter `size`, which equals the vector dimension.

- `double scalar_prod(double vector1[], double vector2[], int size)`
  - This function performs the Vector Scalar Product (a.k.a. dot product) of the two vectors.
  - The function returns the scalar product of the two input vectors, `vector1` and `vector2`.
  - You can assume that both vectors have the same dimension defined by `size`.

- `double norm2(double vector1[], int size)`
  - This function performs the Vector Norm2 (a.k.a. scalar norm) of a vector.
  - The function returns the norm2 value of the input vector, `vector1`.

Add minimally 3 more test cases **for each function** to ensure their functionalities are correctly implemented. Think about some ***Edge Cases*** where an incomplete algorithm might fail.

Example of an Edge Case

Given a SquareRoot(float a) function, it will fail if a < 0. You should test such an input and make sure the function rejects the input instead of putting out garbage results.

Tips

- When you pass an array (NOT a string) into a function, you always need to pass its size into the function as well.
- Vector Algebra Review
  - Given two vectors of the same length ***n***: $x = [x_1, x_2, \dots, x_n]$ and $y = [y_1, y_2, \dots, y_n]$
  - Vector Sum: $z = x + y = [\,(x_1 + y_1), (x_2 + y_2), \dots, (x_n + y_n)\,]$
  - Vector Scalar Product: $z = x \cdot y = x_1y_1 + x_2y_2 + x_3y_3 + \dots + x_ny_n$
  - Vector Norm2: $|x| = sqrt(x \cdot x)$

Marking Scheme

- Code Implementation
  - **[1 mark]** Correct implementation of the Vector Sum function.
  - **[1 mark]** Correct implementation of the Vector Scalar Product function.
  - **[1 mark]** Correct implementation of the Vector Norm2 function.

- Code Behaviour, Analysis, and Test Plans
  - **[1.5 mark, 0.5 mark per function]** Correct Program Behaviour – Passing all test cases.
  - **[1.5 mark, 0.5 mark per function]** Additional Test Cases, minimally 3 for each function.

Lab Question 2 – Unusual Matrix Traversal Method [6 marks]

Complete the function **`void`** `diag_scan(`**`int`** `mat[][N3], `**`int`** `arr[])` in Question2.c, where `mat[][]` is the incoming 2-dimensional square matrix, and `arr[]` is a 1-dimensional vector containing the matrix traversal scanning result. `N3` is a preprocessor constant defined in Questions.h.

The function constructs the array arr[] with all elements of the square matrix mat[][] visited in an anti-diagonal scanning order starting at the **top left corner (index 0,0), but in top-right to bottom-left scanning direction**. Refer to the example below for understanding the output order in a more precise manner:

$$mat[][] = \begin{bmatrix} 1 & 12 & 13 & 49 \\ 5 & 16 & 17 & 81 \\ 9 & 10 & 11 & 20 \\ 2 & 45 & 19 & 14 \end{bmatrix}$$

The resultant output should be $arr[] = [1\ \ 12\ \ 5\ \ 13\ \ 16\ \ 9\ \ 49\ \ 17\ \ 10\ \ 2\ \ 81\ \ 11\ \ 45\ \ 20\ \ 19\ \ 14]$

Add minimally 3 more test cases for the function to ensure its functionalities are correctly implemented. Think about some ***Edge Cases*** where an incomplete algorithm might fail.

Marking Scheme

- Code Implementation
  - **[2 marks]** Correct implementation of the scanning algorithm.
  - **[2 marks]** Correct implementation of the output array construction algorithm.
- Code Behaviour, Analysis, and Test Plans
  - **[1 mark]** Correct Program Behaviour – Passing all test cases.
  - **[1 mark]** Additional Test Cases, minimally 3.

Lab Question 3 – Efficient Vector Representation using Struct [8 marks]

A sparse vector is a vector whose most components are zero. To store a sparse vector efficiently, it is enough to store only its non-zero components and their indices (position in the vector). The components of a vector are indexed starting from 0, like in C arrays. More specifically, in order to store a sparse vector with **n** components, but only **k** of which are non-zero, we can use an array of structs as defined below, where `val` stores the non-zero value, and `pos` stores the corresponding index at which the non-zero value is located in the vector.

```
struct Q3Struct
{
      int val;
      int pos;
};
```

This way, you can use an array of `Q3Struct` to create an efficient representation of a sparse vector as illustrated below:

- Given **`int`** `vector[8] = {0, 0, 23, 0, -7, 0, 0, 48}` (**n** = 8)
- We can infer **k** = 3 because of only three non-zero values in the vector.
- And the efficient representation of this sparse vector would be:
  - **`struct`** `Q3Struct effVector[3] = { {23, 2}, {-7, 4}, {48, 7} }`
  - Or, in tabulated format of what's in `effVector[3]`:

	effVector[0]	**effVector[1]**	**effVector[2]**
val	23	-7	48
pos	2	4	7

  - Notice that the **pos** data member of each element in the struct array is in increasing order.
  - **We will assume that each incoming sparse vector contains at least one non-zero element.**

Complete the function `efficient()` with prototype provided in Question3.c

- **`void`** `efficient(`**`const int`** `source[],` **`struct`** `Q3Struct effVector[],` **`int`** `size)`

which computes the efficient representation of vector source by filling the `Q3Struct` array, `effVector[]`, with the non-zero `val` and their corresponding `pos`. Parameter `size` represents the size of the input array `source[]`. You may assume the size of the `Q3Struct` array equals the number of non-zero values of the vector source.

Complete the function `reconstruct()` with prototype provided in Question3.c

- **`void`** `reconstruct(`**`int`** `source[],` **`int`** `m,` **`const struct`** `Q3Struct effVector[],` **`int`** `n)`

which reconstructs the sparse vector, `source[]`, from the efficient representation stored in the `Q3Struct` array, `effVector[]`, in the format of `val` and `pos`. Parameter `m` represents the size of the array `source[]`, which equals to the dimension of the sparse vector. Parameter `n` is the size of the array `effVector[]`.

Add minimally 2 more test cases **for each function** to ensure their functionalities are correctly implemented. Think about some ***Edge Cases*** where an incomplete algorithm might fail.

Marking Scheme

- Code Implementation
    - **[3 marks]** Correct implementation of the efficient() function.
    - **[3 marks]** Correct implementation of the reconstruct() function.
- Code Behaviour, Analysis, and Test Plans
    - **[1 mark]** Correct Program Behaviour – Passing all test cases.
    - **[1 mark]** Additional Test Cases, minimally 2 combined cases – testing both efficient and reconstruct.

Lab Question 4 – Struct and Sorting [10 Marks]

Recall ***Bubble Sort*** from Lab 1 Question 4. Bubble sort is one of the simplest general-purpose sorting algorithms, implying that it is not limited to sorting integer arrays only.

A struct defined for this question in Questions.h is shown below:

```
struct Q4Struct
{
	int intData;
	char charData;
};
```

The first part of Lab 2 Question 4 contains a ***faulty*** implementation of a Bubble Sort function to sort an array of struct `Q4Struct` in **Ascending Order using the `intData` member as the sorting reference**. Use the VSCode IDE Debugger as instructed in class to fix the implementation, pass all the test cases, and create a debugging report. There are a total of 2 semantic (behavioural) bugs.

Debugging Report Requirements

- Must set up relevant breakpoints and watched variables around the suspected buggy code section.
- Take screenshots of the exact moment where the debugger captures the incorrect program behaviour by using debugger stepping.
- Show how your proposed bug-fix eliminates the incorrect program behaviour with debugger output evidence.
- Even if you catch the bug by inspection / by code walkthrough, you still must carry out the aforementioned debugging process. We will not accept a debugging report without debugger output evidence.

The second part of Lab 2 Question 4 requires you to implement an alternative general purpose sorting algorithm called **Selection Sort** to sort the same input `Q4Struct` array in ***Ascending Order*** **using the `intData` member as the sorting reference**.

- `void sortDataBySelection(struct Q4Struct array[], int size)`

The general algorithm of selection sort is described below, and the graphical representation is provided at the end of the lab manual to help you understand selection sort in better detail.

1. From the start of the data array, visit each index only once.
2. For each visited element at index ***i*** (current element), compare its value against the values of every subsequent element whose index is greater than ***i***. For Ascending Order sorting, find the element with the smallest value (target element), and record its index.
3. If current element and target element do not share the same index (i.e., the smallest element is not the visited element), swap the current element with the target element.
   This is considered completing 1 sorting pass.
4. Move on to the next index, repeat Step 2-3.
5. The sorting process is done when reaching the last element of the array.

The benefit of selection sort over bubble sort is its deterministic nature – selection guarantees that for all arrays of the same size, the number of elements visited will be identical. It also guarantees that the number of swaps done will be capped at the size of the array. However, as you will learn in COMPENG 2SI3, this benefit is not significant when the array size becomes large. That said, selection sort is still a popular general purpose simple sorting algorithm when an approximately constant sorting time is sought after.

Watch Out! When swapping two structs, all data members must be swapped. Never swap only part of the struct – you will destroy the data integrity of the structs.

Add minimally 1 more test case **for each function** to ensure their functionalities are correctly implemented. Think about some ***Edge Cases*** where an incomplete algorithm might fail.

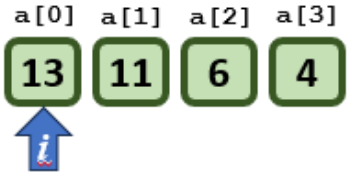
For your workload management, **DO NOT** try to come up with an array size larger than 10.

Marking Scheme

- Code Implementation
  - **[1 marks]** Repair the implementation of Bubble Sort with Q4Struct array.
  - **[1 marks]** Correct implementation of minima-searching algorithm in Selection Sort.
  - **[1 mark]** Correct implementation of struct element swapping in Selection Sort.
  - **[1 mark]** Correct array iteration and stopping condition for Selection Sort.
- Code Behaviour, Analysis, and Test Plans
  - **[1 mark]** Correct Program Behaviour – Passing all test cases.
  - **[4 marks, breakdown below]** Debugger Report
    - **For each of the two bugs**
      - **[0.5 mark]** Setting up effective breakpoints.
      - **[0.5 mark]** Setting up effective variable watches.
      - **[0.5 mark]** Showing evidence of the buggy behaviour.
      - **[0.5 mark]** Showing evidence of the fixed behaviour.
  - **[1 mark]** Additional Test Cases, minimally 1 for each sorting algorithm.

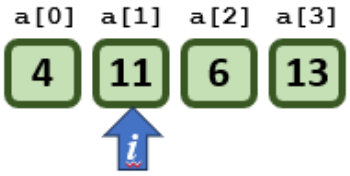
Graphical Representation of Selection Sort

Pass 1



@i=0
Q: Smallest value from a[0] to a[3]?
A: a[3]=4 (i=3)
Action: Swap a[3] and a[0]

Pass 2

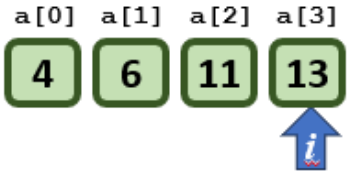


@i=1
Q: Smallest value from a[1] to a[3]?
A: a[2]=6 (i=2)
Action: Swap a[2] and a[1]

Pass 3

@i=2
Q: Smallest value from a[2] to a[3]?
A: a[2]=11 (i=2)
Action: No Swap (Smallest is itself)

Pass 4



@i=3, SORTING DONE