Candidate Number: 279145

Programming for Engineers – H1038

University of Sussex

Assignment 01

Programming for Engineers

Word count: 2495 excluding in-text code and appendix

Table of Contents:

Introduction:

The assignment has provided us with a set of questions and problems we had to solve using C. Below is the condensed code summary that will in essence outline the work that has been done on this problem set.

Requirement specification:

1. **User Input Value assignment**:

This part of the code prompts the user to input their Candidate Number. Then the program writes the 6-digit number as individual variables that will be later used in calculations.

2. **Interactive Menu System**:

After scanning the number, a menu pops up, providing the user with options which will perform certain calculations and operations which will be outlined later in the report.

3. **Options with individual functions for calculations to be carried out**

After displaying the menu system for the user, the program scans the user's choice of an option with the corresponding function. After the user has chosen their option of choice, a series of calculations and/or calculations and analysis is carried out to calculate the necessary values.

Main function detailed overview:

The main function is a function that sets up an interactive menu from which a user can input their 6-digit candidate number and have it written into memory as 6 individual variables that will enable calculations in the functions which the user will choose after entering their CandNo. Below is a step-by-step explanation of the code:

1. Variable Definition:

The code begins with defining of three types of variable types: double, double complex and int. The “double” datatype is used for ease of complex calculations which will result in a non-whole number. If “int” were to be used for these variables, the values would be rounded and would not represent the actual calculated value. The choice of “double complex” variables is mainly since the function that finds the roots of a quadratic equation, might return a square root of a negative number which would not be possible and would return “nan” as the output for this variable. This will be further explored in Option 3 overview. “Int” variables are only the variables that will not undergo any editing to them.

2. Input Candidate Number:

The program asks the user to input their six-digit candidate number, which will be used for calculations. This input is stored in the “cand_no_debug” integer variable.

3. Parsing Candidate Number:

The six digits of the candidate number are individually extracted by dividing and then subtracting the multiples of powers of 10. These individual digits are assigned to variables `s0` through `s5`. It is done using the rounding feature of “int” datatype however it could also have been done using modulo.

4. Interactive Menu:

After the user ha input their CandNo, an interactive menu is presented with options 1 to 5 and 9, which correspond to various calculations and functions. The user is prompted to enter their choice of option.

5. Switch Statement:

The program uses a “switch” statement to handle the user's choice. Each option corresponds to a case in the switch:

- Case 1: Calls “option_1_calculation” function to calculate previously undefined variables `s6`, `s7`, and `s8`.
- Case 2: Calls “option_2_calculation” for vector and angle-related calculations.
- Case 3: Calls “option_3_calculation” to compute the roots of a quadratic function.
- Case 4: Calls “option_4_calculation” for matrix cell substitution calculations.
- Case 5: Calls “file_read_analysis” to read from and analyze a file.
- Case 9: Exits the program.

6. Looping Menu:

A ‘do-while’ loop makes sure that the menu does not disappear after an option is selected, allowing the user to perform all functions within one run of the program.

7. Exiting the Program:

This option closes the program on the user’s request.

Option 1:

Process:

This option is responsible for calculating the variables s6, s7, and s8.

The mathematical operations that are performed are listed below:

$$s6 = \frac{(s0 + s1)}{3}$$

$$s7 = \frac{(s0 + s1 + s2 + s3 + s4 + s5)}{6}$$

$$s8 = s5^{1.25}$$

The following mathematical operations are done via the help of "math.h" library that allows for additional mathematical operations outside of "stdio.h" library and the base capabilities of C. The code for this is provided below:

```
    *s6 = (s0 + s1) / 3; // SIMPLE ALGEBRA
    *s7 = (s0 + s1 + s2 + s3 + s4 + s5) / 6; // AVERAGE
    *s8 = pow(s5, 1.25); // MATH.H POWER FUNCTION
```

The use of the "double" variable type is justified by the new variables will most likely be non-whole numbers therefore "int" would not be suitable.

These variables are then stored in program's memory for other functions to utilize them in their own processes.

Terminal output:

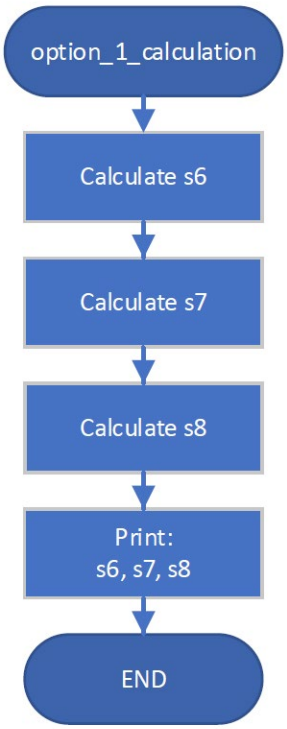
```
Input your six-digit Candidate Number: 279145

Interactive Menu:
1. Additional variable calculation
2. Vector calculation
3. Quadratic root calculation
4. Matrix calculation
5. File read analyze
9. Exit
OptionNo: 1

s6 = 3.000000
s7 = 4.666667
s8 = 7.476744
```

After the user inputs their Candidate Number the mathematical operations as seen above are processed. The updated variables are:

S6 = 3.000
S7 = 4.667
S8 = 7.477

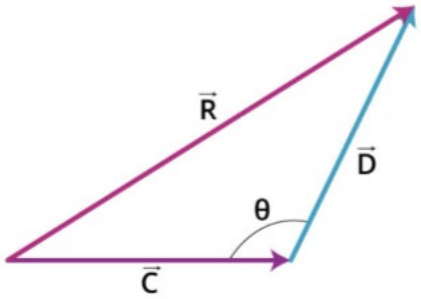
These numbers align with calculations carried out independently purely for verification of methods validity.

Flowchart:



Option 2:

This function creates 2 vectors – C and R with an angle theta in between them. Later a vector R is formed between the beginning of vector C and the end of the vector D. Magnitude of vector C is s0 variable and magnitude of vector D is s1. The diagram for this problem is provided below:



The problem asks us to find the magnitude of vector R and the angle (alpha) between vectors C and R. However, since we do not have coordinates of the vectors, we only have their magnitude. This is convenient as trigonometric formulas can be used to find magnitude of R and alpha. The value of theta is also calculated from variables s0 and s1.

$$\theta = 8 \times s0 \times s1$$

However, this calculation gives us the angle in degrees and the cosine function works with values in radians. Therefore, an additional operation must be done.

$$\theta_{rad} = \theta_{deg} \times \frac{\pi}{180}$$

Another concern is that if the angle is above 360 degrees or 2 π, the calculation might give off a wrong value. Theoretically the maximum value of theta is 648 degrees. Therefore, if theta is above 360 degrees of 2 π, we have to make the following calculation that provides us an angle value that can be worked with.

Code for this conversion and calculation can be seen below:

```
vector_c = s0;
    vector_d = s1;
    theta = (8 * s0 * s1);
    theta = theta * (pi/180); // CONVERSION TO RADIANS
    if(theta > 2*pi)          // IN CASE ANGLE IS ABOVE 2PI (360 DEGREES)
    {
        theta = theta - (2*pi); // NEW THETA IF ANGLE IS TOO HIGH
    }
```

Mathematical formulas can be seen below:

$$R = \sqrt{C^2 + D^2 - (2)(C)(D)\cos\theta}$$

Code for this can be seen below:

```
intermediate_vector_variable = pow(vector_c, 2) + pow(vector_d, 2) - (2 * vector_c
* vector_d * cos(theta));
vector_r = sqrt(intermediate_vector_variable);
```

Intermediate vector variable is placed there to make the code look better and for me not to get confused with mathematical operations. This intermediate vector variable represents R^2 which will then be taken the root of, resulting in magnitude of R, which allows us to go onto further calculations.

This then allows us to find the magnitude of vector R, therefore allowing us to now use the sine formula to calculate alpha. The mathematical formula can be seen below:

$$\frac{\sin\alpha}{D} = \frac{\sin\theta}{R};\ \alpha = \sin^{-1}\frac{\sin\theta \times D}{R}$$

The code for this mathematical operation can be seen below:

```
alpha = asin((sin(theta) * vector_d) / vector_r); // DETERMINING ALPHA (ANGLE
BETWEEN R AND C)
alpha = (alpha * 180) / pi;
```

Additionally, we must convert it back to degrees by multiplying alpha by 180 / π. These values are then printed out for user to confirm and use.

Terminal output:

```
1. Additional variable calculation
2. Vector calculation
3. Quadratic root calculation
4. Matrix calculation
5. File read analyze
9. Exit
OptionNo: 2

Magnitude of R = 7.967904
alpha = 54.546759
```

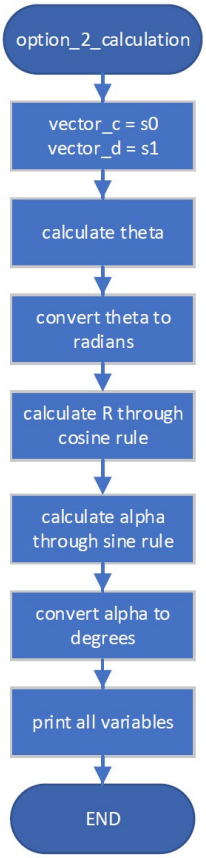
After the user inputs their Candidate Number the mathematical operations as seen above are processed. The updated variables are:

$$R = 7.967904$$

$$\alpha = 54.546759$$

These numbers align with calculations carried out independently purely for verification of methods validity.

Flowchart:



Option 3:

This function creates a quadratic equation based on variables s0, s1, s2. The function then finds two roots of the quadratic equation. The variables s0, s1 and s2 act as a, b and c parameters of the quadratic equation. The mathematical equation can be seen below:

$$0 = ax^2 + bx + c;\ \ where\ a = s0, b = s1, c = s2$$

The proper way of finding roots of a quadratic equation can be described by a formula that can be seen below:

$$x = \frac{-b \pm \sqrt{b^2 - 4ac}}{2a} = \frac{-(s1) \pm \sqrt{(s1)^2 - 4(s0)(s2)}}{2(s0)}$$

This can be done via the use of "math.h" library, the code for which looks like:

```
discriminant = s1 * s1- 4 * s0 * s2;
    if(discriminant >= 0) // REAL ROOTS
    {
        *quadratic_root_1 = (-s1 + sqrt(discriminant)) / (2 * s0);
        *quadratic_root_2 = (-s1 - sqrt(discriminant)) / (2 * s0);
    }
    else // COMPLEX ROOTS
    {
        double realPart = -s1 / (2*s0);
        double imagPart = sqrt(-discriminant) / (2*s0);
        *quadratic_root_1 = realPart + imagPart * I;
        *quadratic_root_2 = realPart - imagPart * I;
    }
```

However, there was an issue that required further work and research. Since "math.h" and C's native abilities are unable to process complex numbers, a new library had to be added to the code. "Complex.h" is a library that can process square roots of complex numbers, allowing the code to find all roots, complex included. As can be seen in the code seen above, first the program calculates the discriminant, and if the discriminant is negative, then the function splits the quadratic equation formula gets split into real part and complex part, calculated separately. Then the function writes the values of real part of the roots and the imaginary part of the roots and is then pointed at for the main function to be able to print the result out.

```
printf("\nRoot 1: %.2f%+.2fi\n", creal(quadratic_root_1),cimag(quadratic_root_1));
printf("Root 2: %.2f%+.2fi\n", creal(quadratic_root_2), cimag(quadratic_root_2));
```

Terminal output:

```
Input your six-digit Candidate Number: 279145

Interactive Menu:
1. Additional variable calculation
2. Vector calculation
3. Quadratic root calculation
4. Matrix calculation
5. File read analyze
9. Exit
OptionNo: 3

Root 1: -1.75+1.20i
Root 2: -1.75-1.20i
```

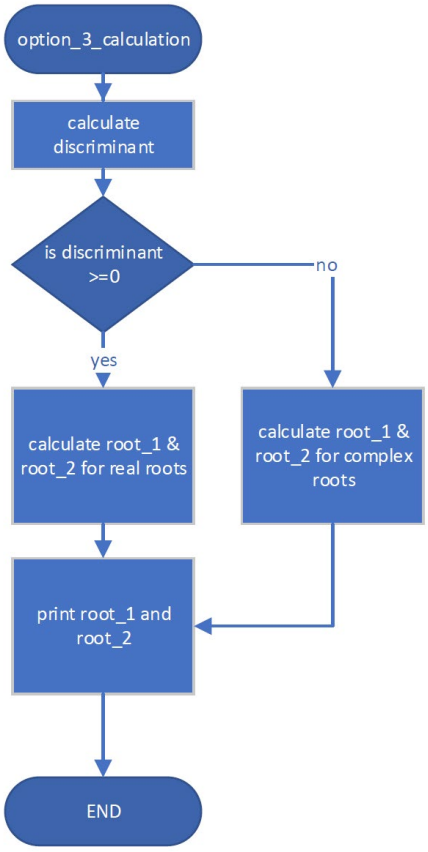
After the user inputs their Candidate Number the mathematical operations as seen above are processed. The updated variables are:

$$\text{Root 1} = -1.75 + 1.20i$$

$$\text{Root 2} = -1.75 - 1.20i$$

These numbers align with calculations carried out independently purely for verification of methods validity.

Flowchart:



Option 4:

This function takes the user input variables s0 through s5, as well as the calculated variables s6, s7 and s8, writes them into an array and displays it in form of a 3x3 matrix. This is done through a "for" loop, the code for which can be seen below:

```
int i = 0;
    for (i ; i < 3; i++)
    {
        int j = 0;
        for (j ; j < 3; j++)
        {
            printf("%lf ", arr[i][j]);
        }
        printf("\n");
    }
```

The loop prints out array values 3 values at a time, after which it is transferred to the next line to create a matrix of the required format.

Then the function prompts the user to choose a row and column and a value to overwrite the chosen cell. Before overwriting, the function verifies that the user's choice of row and column are within the boundaries of the matrix. I.E., if the user wants to overwrite column [5] or row [-999] the function does not allow that and ends the function before prompting the user to choose a different function or to have another attempt at choosing an existing column. The code for this part can be seen below:

```
if (user_row < 3 && user_row >= 0 && user_column < 3 && user_column >= 0) {
        double *ptr = &arr[0][0];

*(ptr + user_column * 3 + user_row) = new_userdef_variable;        //  UPDATE THE
CHOSEN ELEMENT USING POINTER ARITHMETIC
```

If the user chooses a valid cell to overwrite with a new numeric value, the function calculates the correct position in the memory where the chosen element lies using pointer arithmetic and updates the value at that position with the new value provided by the user. The function uses pointers to reach first function's calculated values s6, s7, s8, without having to perform the calculations on its own.

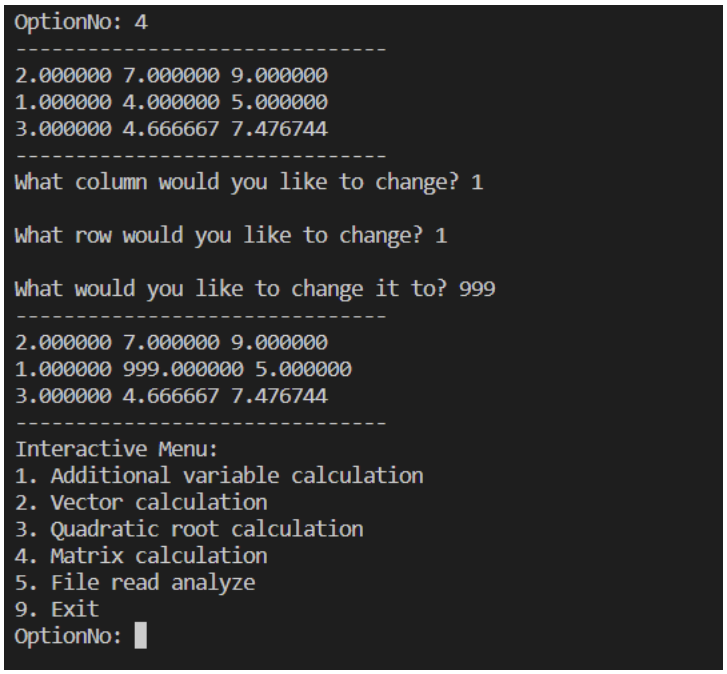
After which the updated matrix is printed out, showing the overwritten value in the matrix. Code for this can be found below:

```
    int ii = 0;
    for (ii ; ii < 3; ii++)
    {
        int jj = 0;
        for (jj ; jj < 3; jj++)
        {
            printf("%lf ", arr[ii][jj]);
        }
        printf("\n");
    }
```

Counting variable from the original loop cannot be utilized here due to compiler limitations, therefore a substitute in form of "ii" and "jj" is used for counting cycles of the "for" loop.

The function then returns [0] to indicate successful execution of the function.

Terminal output:

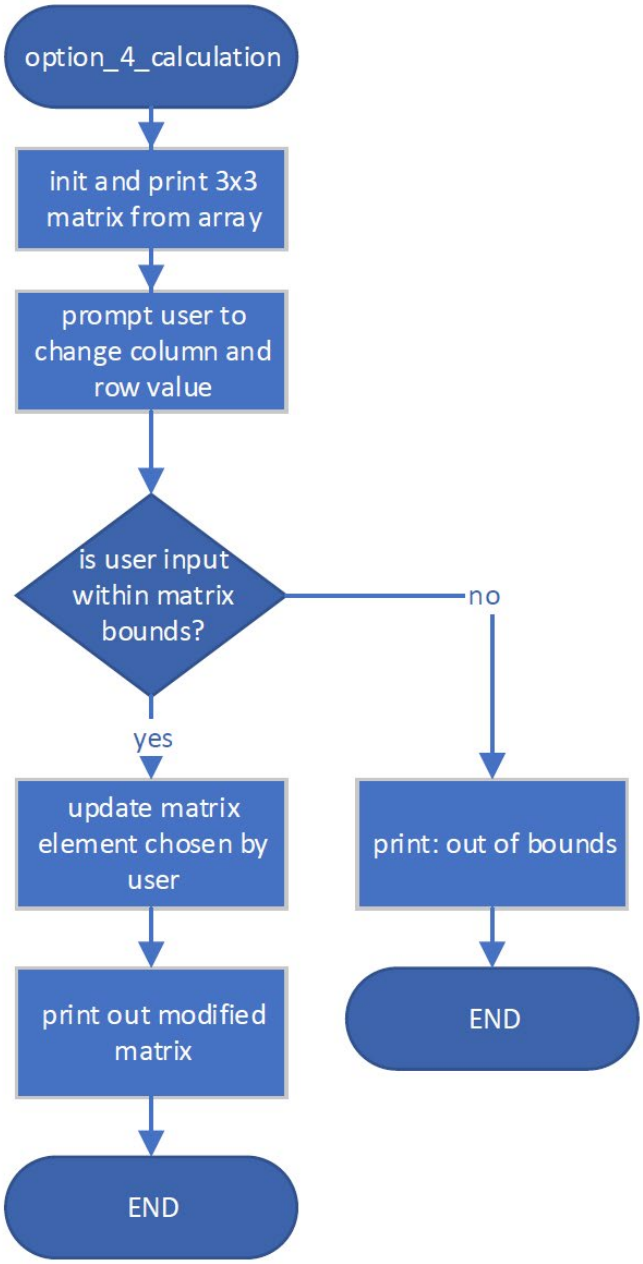


After the user inputs their Candidate Number the mathematical operations as seen above are processed. The updated variables are:

Matrix cell [1][1] is overwritten with [999]

the success of which is reprinted in another matrix as seen in the terminal output above.

Flowchart:


option_4_calculation
init and print 3x3 matrix from array
prompt user to change column and row value
is user input within matrix bounds?
no
yes
update matrix element chosen by user
print: out of bounds
print out modified matrix
END
END

Option 5:

As per assignment, this function is supposed to be computed independently of user's choice, however if the function would be executed regardless of user's choice, any error with reading and/or locating the file would stop the program from computing all other options. Another reason for this is to not clutter the terminal window with data the user might not be aware of, therefore this function is considered a function to be chosen by the user to execute from the options menu.

This function reads the data file called data.txt and writes the values into an array to be used in further calculations. Initially the function has to open the file from the same directory as the file and write the values from the list into the array titled "arr". Code for this action can be seen below:

```
FILE *stream;
    int count = 0, arr[21], sum = 0, pos_nums = 0, odd_nums = 0, negative_sum = 0,
negative_sum_cubed = 0;

    stream = fopen(DATAFILE, "r");
    do
    {
        fscanf(stream, "%d", &arr[count]);
        sum = sum + arr[count];
        printf("%d\n", arr[count]);
        count++;
    }
    while ((fgetc(stream) != EOF) && (count < (sizeof(arr)/sizeof(arr[0]))));
    fclose(stream);
```

This part of the code writes the data via a loop "while" where for the amount of array spaces defined prior to the program in "arr[21]" allowing only 22 spaces as that is the amount of values in the data.txt file, not to force the function to scan and analyze empty spaces in the array in further calculations. The "while" loop compares the "count" to array spaces, adding "1" to count every loop, until it reaches the value of [21], at which the function stops scanning and proceeds to further calculations.

If the function is unable to locate or open the file, the function returns "NULL" and is then notifying the user of the location error, ending the function, allowing the user to choose another option or close the program to verify data.txt file placement. Code for which can be seen below:

```
if (DATAFILE == NULL) {
        printf("Error opening file");
        return 1;
    }
```

When the file has been successfully opened, and all the values have been recorded into the array the function may continue to carry out necessary calculations. The first calculation/analysis that the function carries out is finding out a value that is the largest in the list. There are a lot of sorting algorithms to do this, however this function scans every next value comparing it to the previously “max” value. If the program detects a value that is larger than the previously "max" value, the “max" value is overwritten with the new largest value. This loop goes through the entire array and therefore figures out the largest value in the entire array. The code for this sorting algorithm can be seen below:

```
int max = arr[0]; // ASSUME FIRST ELEMENT TO BE THE MAXIMUM
// FIRST ELEMENT ALREADY DEFINED THEREFORE START FROM THE SECOND ELEMENT
    int i = 1;
    for (i ; i < 21; ++i)
    {
        if (arr[i] > max)
        {
            max = arr[i]; // Update max when a larger value is found
        }
    }
```

After this part of the function is done, the function now has to find out the number of positive values as well as the number of odd values in the array. This is all done through a similar loop which scans the entire array, comparing the current value against a condition, and if the value satisfies the condition, a counting variable is then increased by one. The condition for a value to qualify as a positive is that it must be above zero. The condition for a value to be considered odd is to be divisible by 2 with a remainder and not be equal to zero.

The code for this sorting algorithm can be seen below:

```
int iii = 0;
    for (iii ; iii < 21; ++iii) {
        if (arr[iii] > 0) {
            pos_nums++; // Increment when value is positive
        }
        if (arr[iii] % 2 != 0) {
            odd_nums++; // Increment when value is odd
        }
        if (arr[iii] < 0)
        {
            negative_sum += arr[iii]; // Add the value to sum of negative numbers
        }
    }
```

The loop continues scanning until the variable “iii” reaches the value of 21, at which point the list ends, therefore not wasting the memory available to this function.

After these calculations have been finished, the function now must find the sum of all negative numbers, and then cube the resulting sum. This can be done either through the use of “math.h” function, or through multiplying three sums together, however the “pow()” function is more efficient at this as it also helps clean up the code in case someone will be analyzing it in the future. For this function to find the sum of all negative values it has another loop, through which it scans the array, and if the value matches the condition of being less than zero, a variable “negative_sum”, initially defined as being zero, is combined with the value that matches the condition. After the entire array of values has been scanned and the sum of all negative values has been found, the function cubes that value to result in overwrite of the variable “negative_sum_cubed”, hence finishing all its conditions, printing out all the results.

The code for the process described can be seen below:

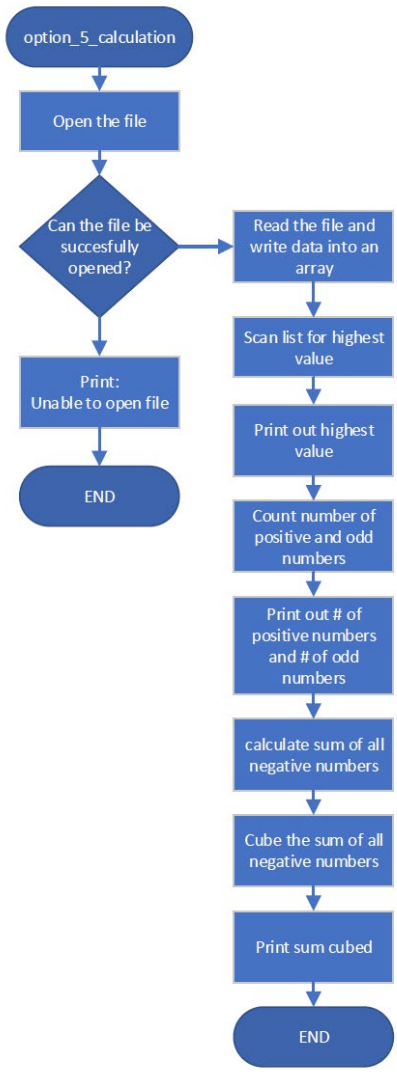
```
int iii = 0;
    for (iii ; iii < 21; ++iii) {
        if (arr[iii] > 0) {
            pos_nums++; // Increment when value is positive
        }
        if (arr[iii] % 2 != 0) {
            odd_nums++; // Increment when value is odd
        }
        if (arr[iii] < 0)
        {
            negative_sum += arr[iii]; // Add the value to sum of negative numbers
        }
    }
    negative_sum_cubed = pow(negative_sum, 3);
```

Terminal output:

```
OptionNo: 5
0

The maximum value in the array is: 1077704315
The array contains 5 positive values.
The array contains 4 odd values.
Negative sum: -1158343768
Negative sum cubed: -2147483648
```

After the user inputs their choice of option, mathematical operations as seen above are processed.

Flowchart:



Appendix:

Source code:

```
// LIBRARIES AND CONSTANTS
#include <stdio.h>
#include <math.h>
#include <complex.h>
#define pi 3.1415
#define DATAFILE "data.txt"
/*=============================================================================================
=========================================================================*/
// DECLARE FUNCTIONS BEFORE MAIN TO AVOID CONFUSION OF THE COMPILER
int option_1_calculation(double s0, double s1, double s2, double s3, double s4, double s5, double* s6,
double* s7, double* s8);
int option_2_calculation(double s0, double s1, double s2, double s3, double s4, double s5, double s6,
double s7, double s8, double vector_c, double vector_d, double vector_r, double theta, double alpha,
double intermediate_vector_variable);
int option_3_calculation(double s0, double s1, double s2, double s3, double s4, double s5, double s6,
double s7, double s8, double discriminant, double root_intermediate_variable, double complex
*quadratic_root_1, double complex *quadratic_root_2);
int option_4_calculation(double s0, double s1, double s2, double s3, double s4, double s5, double s6,
double s7, double s8);
int file_read_analysis(double s0, double s1, double s2, double s3, double s4, double s5, double s6,
double s7, double s8);
/*=============================================================================================
=========================================================================*/
int main()
{

// DEFINING ALL VARIABLES WITH APPROPRIATE DATA TYPES
    double s0, s1, s2, s3, s4, s5, s6, s7, s8, vector_c, vector_d, vector_r, theta, alpha,
intermediate_vector_variable, discriminant, root_intermediate_variable;
    double complex quadratic_root_1, quadratic_root_2;
    int cand_no_debug, option_choice, option;
/*=============================================================================================
=========================================================================*/
// WRITING THE CANDIDATE NUMBER AS INDIVIDUAL DIGITS FOR FURTHER CALCULATIONS (COULD BE DONE THROUGH
MODULO)
    printf("Input your six-digit Candidate Number: ");
    scanf("%d", &cand_no_debug);

    s0 = cand_no_debug / 100000;
    s1 = (cand_no_debug / 10000) - (s0 * 10);
    s2 = (cand_no_debug / 1000) - (s0 * 100) - (s1 * 10);
    s3 = (cand_no_debug / 100) - (s0 * 1000) - (s1 * 100) - (s2 * 10);
    s4 = (cand_no_debug / 10) - (s0 * 10000) - (s1 * 1000) - (s2 * 100) - (s3 * 10);
    s5 = (cand_no_debug / 1) - (s0 * 100000) - (s1 * 10000) - (s2 * 1000) - (s3 * 100) - (s4 * 10);
/*=============================================================================================
=========================================================================*/
```

```
// INTERACTIVE MENU FOR OPTIONS TEXT
  do {
        printf("\nInteractive Menu:\n");
        printf("1. Additional variable calculation\n");
        printf("2. Vector calculation\n");
        printf("3. Quadratic root calculation\n");
        printf("4. Matrix calculation\n");
        printf("5. File read analyze\n");
        printf("9. Exit\n");
        printf("OptionNo: ");
        scanf("%d", &option);
/*=============================================================================================
=========================================================================*/
// INTERACTIVE MENU OPTIONS CASES
        switch(option) {
// OPTION 1 - DETERMINING S6, S7 AND S8 VARIABLES
            case 1:
                option_1_calculation(s0, s1, s2, s3, s4, s5, &s6, &s7, &s8);
                break;
// OPTION 2 - VECTOR AND ANGLE OPERATIONS
            case 2:
                option_2_calculation(s0, s1, s2, s3, s4, s5, s6, s7, s8, vector_c, vector_d, vector_r,
theta, alpha, intermediate_vector_variable);
                break;
// OPTION 3 - QUADRATIC FUNCTION ROOT CALCULATION
            case 3:
                option_3_calculation(s0, s1, s2, s3, s4, s5, s6, s7, s8, discriminant,
root_intermediate_variable, &quadratic_root_1, &quadratic_root_2);
                    printf("\nRoot 1: %.2f%+.2fi\n", creal(quadratic_root_1),
cimag(quadratic_root_1)); //PRINT GOES OUTSIDE THE FUNCTION DUE TO CLASHES WITH DEFINING THE USED
VARIABLES W/ POINTERS
                    printf("Root 2: %.2f%+.2fi\n", creal(quadratic_root_2), cimag(quadratic_root_2));
//PRINT GOES OUTSIDE THE FUNCTION DUE TO CLASHES WITH DEFINING THE USED VARIABLES W/ POINTERS
                    printf("------------------------------");
                break;
// OPTION 4 - MATRIX CELL SUBSTITUTION
            case 4:
                option_4_calculation(s0, s1, s2, s3, s4, s5, s6, s7, s8);
                break;
// FILE READ AND ANALYSIS, DONE AS A FUNCTION TO MAKE THE CONSOLE CLEANER
            case 5:
                file_read_analysis(s0, s1, s2, s3, s4, s5, s6, s7, s8);
                break;
// EXIT OPTION - CLOSES THE PROGRAM
            case 9:
                printf("Exiting the program.\n");
                break;
// OPTION IN CASE INPUT DOES NOT EQUAL ONE OF THE OPTIONS
```

```
            default:
                printf("Invalid option. Please try again.\n");
        }
    } while(option != 9);
    return 0;
}

/*=========================================================================================================
=========================================================================*/
// FUNCTION FOR OPTION 1
int option_1_calculation(double s0, double s1, double s2, double s3, double s4, double s5, double* s6,
double* s7, double* s8)
{
    *s6 = (s0 + s1) / 3; // SIMPLE ALGEBRA
    *s7 = (s0 + s1 + s2 + s3 + s4 + s5) / 6; // AVERAGE
    *s8 = pow(s5, 1.25); // MATH.H POWER FUNCTION
    printf("\ns6 = %lf \ns7 = %lf \ns8 = %lf\n", *s6, *s7, *s8);
    printf("------------------------------");
    return 0;
}

/*=========================================================================================================
=========================================================================*/
// FUNCTION FOR OPTION 2
int option_2_calculation(double s0, double s1, double s2, double s3, double s4, double s5, double s6,
double s7, double s8, double vector_c, double vector_d, double vector_r, double theta, double alpha,
double intermediate_vector_variable)
{
    vector_c = s0;
    vector_d = s1;
    theta = (8 * s0 * s1);
    theta = theta * (pi/180); // CONVERSION TO RADIANS
    if(theta > 2*pi)          // IN CASE ANGLE IS ABOVE 2PI (360 DEGREES)
    {
        theta = theta - (2*pi); // NEW THETA IF ANGLE IS TOO HIGH
    }
    intermediate_vector_variable = pow(vector_c, 2) + pow(vector_d, 2) - (2 * vector_c * vector_d *
cos(theta));
    vector_r = sqrt(intermediate_vector_variable);
    printf("\nMagnitude of R = %lf", vector_r);

    alpha = asin((sin(theta) * vector_d) / vector_r); // DETERMINING ALPHA (ANGLE BETWEEN R AND C)
    alpha = (alpha * 180) / pi;
    printf("\nalpha = %lf\n", alpha);
    printf("------------------------------");
    return 0;
}
```

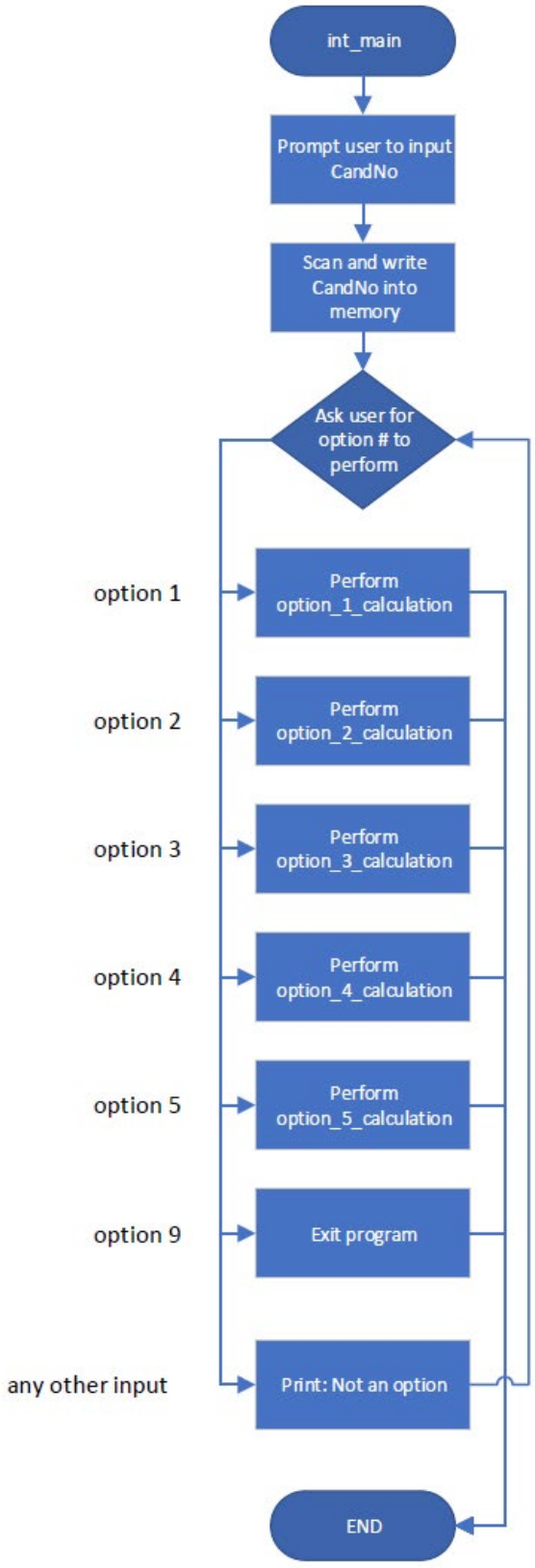
```
/*=======================================================================================================
=========================================================================*/
// FUNCTION FOR OPTION 3
int option_3_calculation(double s0, double s1, double s2, double s3, double s4, double s5, double s6,
double s7, double s8, double discriminant, double root_intermediate_variable, double complex
*quadratic_root_1, double complex *quadratic_root_2)
// UTILIZING THE COMPLEX.H LIBRARY IN ORDER TO OBTAIN ROOTS OF THE QUADRATIC EQUATION EVEN IF THE
DISCRIMINANT IS NEGATIVE WHICH RESULTS IN A SQUARE ROOT OF A NEGATIVE NUMBER REQUIRING
{
    discriminant = s1 * s1- 4 * s0 * s2;
    if(discriminant >= 0) // REAL ROOTS
    {
        *quadratic_root_1 = (-s1 + sqrt(discriminant)) / (2 * s0);
        *quadratic_root_2 = (-s1 - sqrt(discriminant)) / (2 * s0);
    }
    else // OTHER KINDS OF ROOTS (UNREAL/ SINGLE ROOT)
    {
        double realPart = -s1 / (2*s0);
        double imagPart = sqrt(-discriminant) / (2*s0);
        *quadratic_root_1 = realPart + imagPart * I;
        *quadratic_root_2 = realPart - imagPart * I;
    }
    return 0;
}

/*=======================================================================================================
=========================================================================*/
// FUNCTION FOR OPTION 4
int option_4_calculation(double s0, double s1, double s2, double s3, double s4, double s5, double s6,
double s7, double s8)
{
// VARIABLE DEFINITION
    int user_row, user_column;
    double new_userdef_variable;
    double arr[3][3] = {{s0, s1, s2}, {s3, s4, s5}, {s6, s7, s8}};
// CONVERTING ARRAY INTO A 3X3 MATRIX
    printf("------------------------------\n");
    int i = 0;
    for (i ; i < 3; i++)
    {
        int j = 0;
        for (j ; j < 3; j++)
        {
            printf("%lf ", arr[i][j]);
        }
        printf("\n");
    }
    printf("------------------------------\n");
```

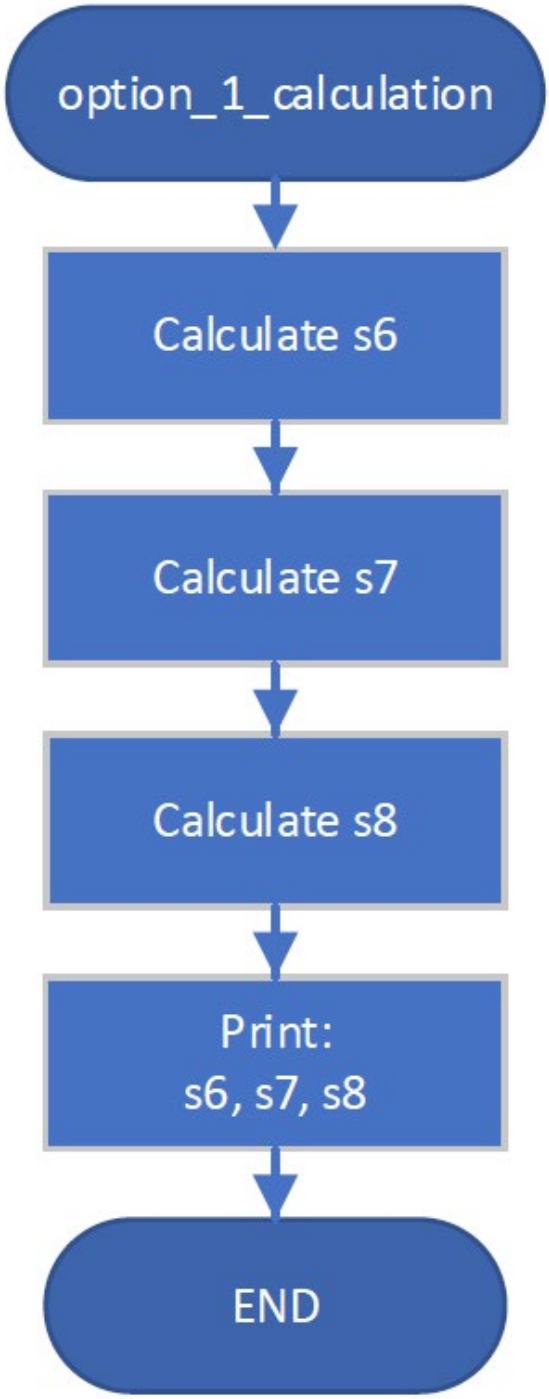
```
// PROMPTING USER TO CHANGE COLUMN AND ROW AND
    printf("What column would you like to change? ");
    scanf("%d", &user_column);
    printf("\nWhat row would you like to change? ");
    scanf("%d", &user_row);
    printf("\nWhat would you like to change it to? ");
    scanf("%lf", &new_userdef_variable);
    printf("------------------------------\n");

// Check that the user's specified indices are within the bounds of the 3x3 matrix.
    if (user_row < 3 && user_row >= 0 && user_column < 3 && user_column >= 0) {
        double *ptr = &arr[0][0];         //  OBTAIN POINTER TO FIRST ELEMENT IN MATRIX ARRAY
        *(ptr + user_column * 3 + user_row) = new_userdef_variable;        //  UPDATE THE CHOSEN
ELEMENT USING POINTER ARITHMETIC
    }
    else
    {
        printf("Invalid indices provided, nothing changed.\n");
    }
// PRINT OUT THE UPDATED MATRIX
    int ii = 0;
    for (ii ; ii < 3; ii++)
    {
        int jj = 0;
        for (jj ; jj < 3; jj++)
        {
            printf("%lf ", arr[ii][jj]);
        }
        printf("\n");
    }
    printf("------------------------------");

    return 0;
}

/*=============================================================================================
=========================================================================*/
// FUNCTION FOR FILE READ AND FURTHER ANALYSIS
int file_read_analysis(double s0, double s1, double s2, double s3, double s4, double s5, double s6,
double s7, double s8)
{
// OPENING AND WRITING DATA.TXT FILE LIKE IN THE SLIDES SHOWN IN LECTURE
    FILE *stream;
    int count = 0, arr[21], sum = 0, pos_nums = 0, odd_nums = 0, negative_sum = 0, negative_sum_cubed
= 0;

    stream = fopen(DATAFILE, "r");
```

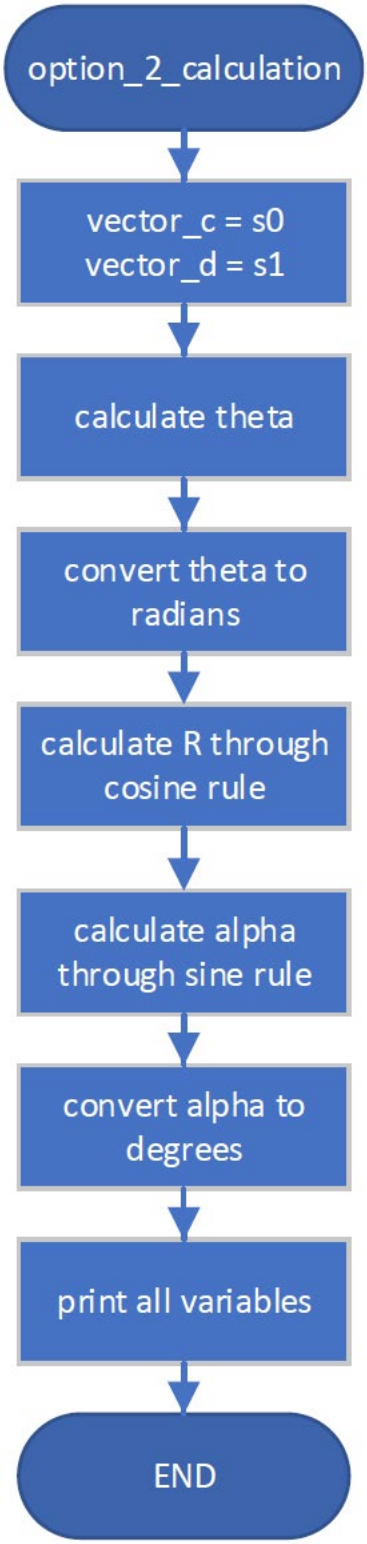
```
    do
    {
        fscanf(stream, "%d", &arr[count]);
        sum = sum + arr[count];
        printf("%d\n", arr[count]);
        count++;
    }
    while ((fgetc(stream) != EOF) && (count < (sizeof(arr)/sizeof(arr[0]))));
    fclose(stream);

    int max = arr[0]; // ASSUME FIRST ELEMENT TO BE THE MAXIMUM
// THE SORTING ALGORITHM USED HERE ASSUMES THAT THE FIRST NUMBER IS THE LARGEST, SCANNING EVERY VALUE
IN THE LIST, LOOKING FOR A NUMBER LARGER THAN THE ONE RECORDED AS THE MAX

// FIRST ELEMENT ALREADY DEFINED THEREFORE START FROM THE SECOND ELEMENT (NOT NECESSARY BUT NICE TO
HAVE)
    int i = 1;
    for (i ; i < 21; ++i)
    {
        if (arr[i] > max)
        {
            max = arr[i]; // Update max when a larger value is found
        }
    }
    if (DATAFILE == NULL) {
        printf("Error opening file");
        return 1;
    }

// LOOP FOR THE IF STATEMENTS TO SCAN ALL THE VALUES IN THE .TXT FILE
    int iii = 0;
    for (iii ; iii < 21; ++iii) {
        if (arr[iii] > 0) {
            pos_nums++; // Increment when value is positive
        }
        if (arr[iii] % 2 != 0) {
            odd_nums++; // Increment when value is odd
        }
        if (arr[iii] < 0)
        {
            negative_sum += arr[iii]; // Add the value to sum of negative numbers
        }
    }
    negative_sum_cubed = pow(negative_sum, 3);

// PRINT OUT ALL THE NECASSARY INFORMATION
    printf("\nThe maximum value in the array is: %d\n", max);
    printf("The array contains %d positive values.\n", pos_nums);
```

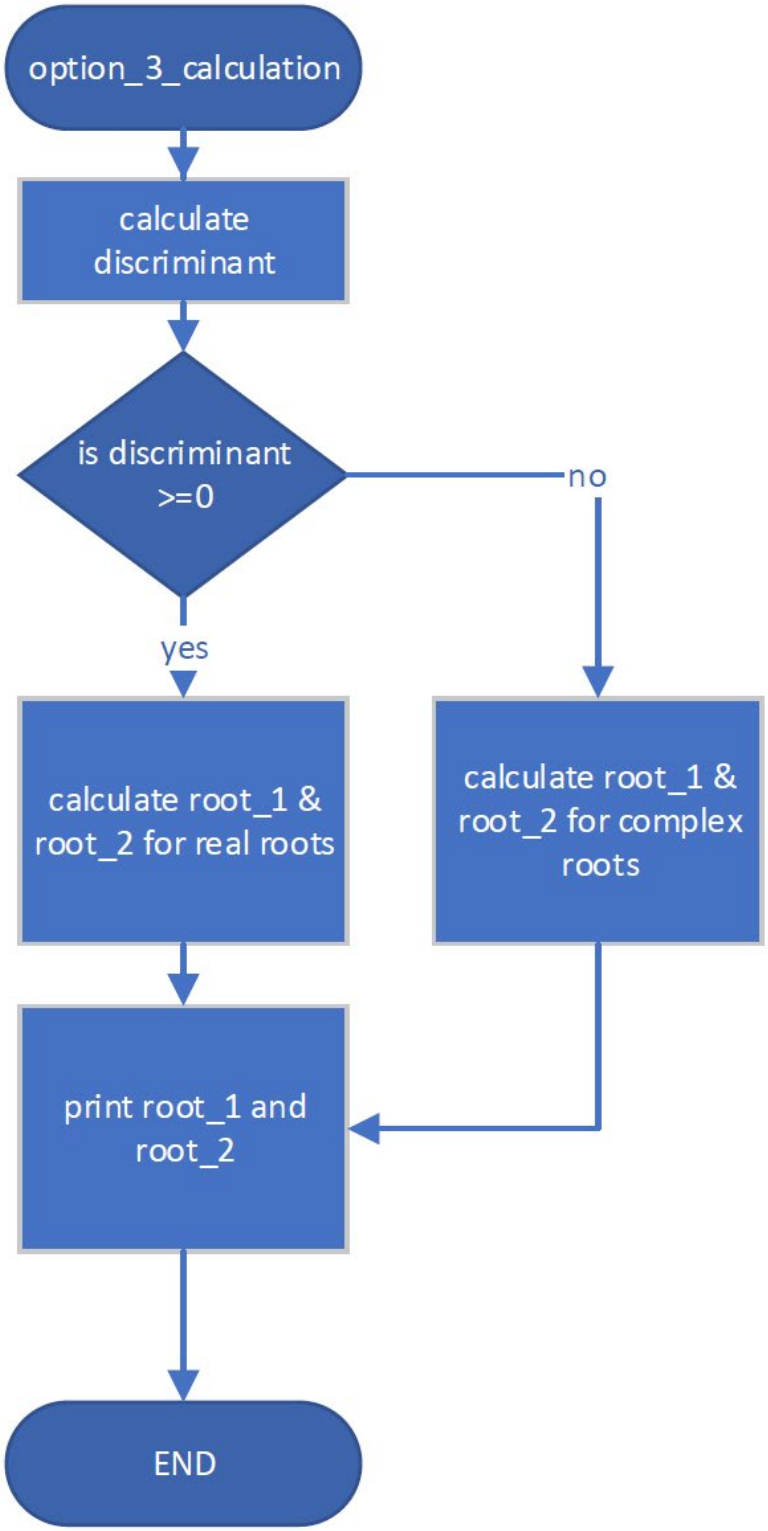
```
    printf("The array contains %d odd values.\n", odd_nums);
    printf("Negative sum: %d\n", negative_sum);
    printf("Negative sum cubed: %d\n", negative_sum_cubed);
    printf("------------------------------");

    return 0;
}
```

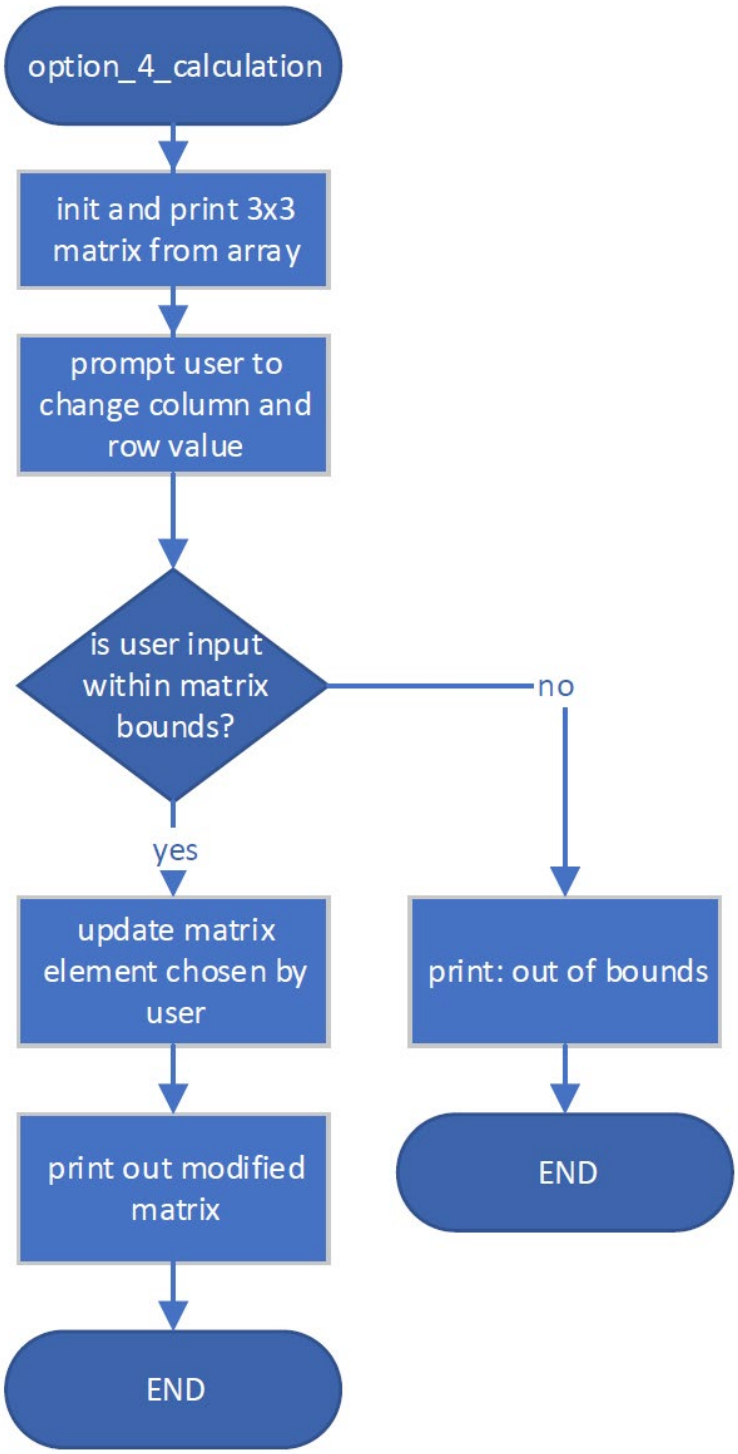
Flowcharts:

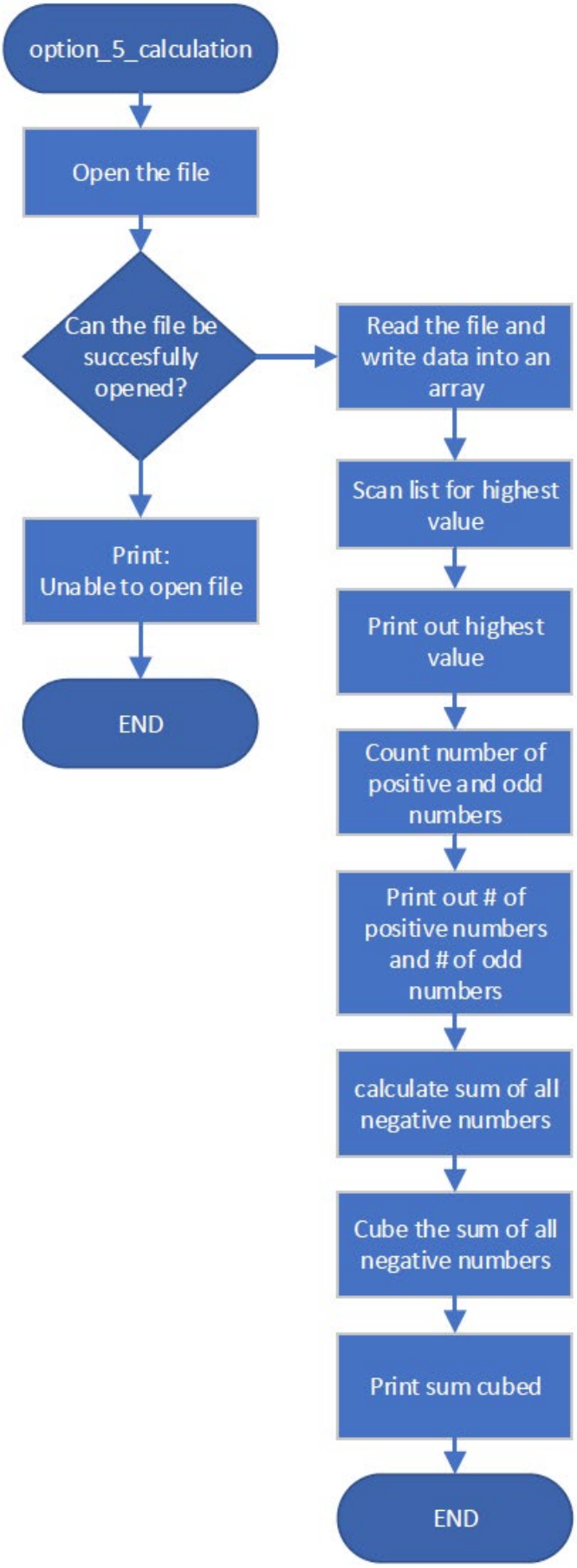

int_main
Prompt user to input CandNo
Scan and write CandNo into memory
Ask user for option # to perform
option 1
Perform option_1_calculation
option 2
Perform option_2_calculation
option 3
Perform option_3_calculation
option 4
Perform option_4_calculation
option 5
Perform option_5_calculation
option 9
Exit program
any other input
Print: Not an option
END


option_1_calculation
Calculate s6
Calculate s7
Calculate s8
Print:
s6, s7, s8
END


option_2_calculation
vector_c = s0
vector_d = s1
calculate theta
convert theta to radians
calculate R through cosine rule
calculate alpha through sine rule
convert alpha to degrees
print all variables
END


option_3_calculation
calculate discriminant
is discriminant >=0
no
yes
calculate root_1 & root_2 for real roots
calculate root_1 & root_2 for complex roots
print root_1 and root_2
END


option_4_calculation
init and print 3x3 matrix from array
prompt user to change column and row value
is user input within matrix bounds?
no
yes
update matrix element chosen by user
print: out of bounds
print out modified matrix
END
END


option_5_calculation
Open the file
Can the file be succesfully opened?
Print: Unable to open file
END
Read the file and write data into an array
Scan list for highest value
Print out highest value
Count number of positive and odd numbers
Print out # of positive numbers and # of odd numbers
calculate sum of all negative numbers
Cube the sum of all negative numbers
Print sum cubed
END