Data Structures & Algorithms Assignment

CSA 1017

247696M

Edward Fleri Soler

Table of Contents

1) Roman Numerals Algorithm

1.1) Source Code

The purpose of this algorithm is to output the Roman numeral equivalent of an integer value inputted by the user. A basic user interface requested the input of an integer between 1 and 1024 (limit may be altered). The algorithm then computed the roman equivalent for the integer, and outputted the results on screen. The following is the implemented source code:

```
#include<stdio.h>
#define SIZE 15

int main(void)
{
    int num, calc;
    num = calc = 0;

    char *ints[] = {"I", "II", "III", "IV", "V", "VI", "VII", "VIII", "IX"};
    char *tens[] = {"X", "XX", "XXX", "XL", "L", "LX", "LXX", "LXXX", "XC"};
    char *hunds[] = {"C", "CC", "CCC", "CD", "D", "DC", "DCC", "DCCC", "CM"};

    printf("Please enter an integer between 1 and 1024: ");
    while((scanf("%d", &num) != 1) || (num < 1) || (num > 1024))
    {
        fflush(stdin);
        printf("\nInvalid Input! - Please enter an integer between 1 and
        1024\n\n");
    }

    if(num >= 1000)
    {
        printf("%c", 'M');
        num -= 1000;
    }

    if(num >= 100)
    {
        calc = num/100;
        printf("%s",hunds[calc-1]);
        //due to integer division, remainder is disregarded

        num -= (calc*100);
    }

    if(num >= 10)
    {
        calc = num/10;
        printf("%s",tens[calc-1]);
        num -= (calc*10);
    }

    if(num >= 1)
    {
        printf("%s",ints[num-1]);
    }

    return 0;
}
```

The above source code firstly declares and initialises three arrays with Roman numeral symbols. The first array corresponds to integers, the second to digits and the third to hundreds. These will later be used to retrieve the desired symbol. The user is then asked to input a positive number within the limit, and a catch is implemented to prevent invalid data being inputted.

The integer is then broken down, having a certain value deducted and replaced by a roman numeral. This is continued until the final Roman numeral equivalent is output on screen.

1.2) Testing

This programs multiple features were tested to ensure that it runs properly. Firstly the user interface and input validation was tested by entering values out of range, such as large values or negative values; as well as characters. Once this test showed no flaws, the Roman numeral algorithm was tested by input various random integer values and observing the output for inconsistencies. The output was evaluated and compared to the expected results, bringing an end to the testing phase.

1.3) Screen Dumps

The following are a number of screen dumps, authenticating the positive test results obtained by this program:

```
Please enter an integer between 1 and 1024: -1

Invalid Input! - Please enter an integer between 1 and 1024

1025

Invalid Input! - Please enter an integer between 1 and 1024
```

```
45

XLV

Please enter an integer between 1 and 1024: 723

DCCXXIII

Please enter an integer between 1 and 1024: 1018

MXVIII
```

As may be seen, the program performs its expected job and succeeds in handling input validation.

2) Reverse Polish Notation Algorithm

2.1) Source Code

The purpose for this algorithm is to evaluate arithmetic expressions in reverse polish notation and to output there resulting value on screen. An added feature includes the output of expressions being evaluated at each step of the process. This program was based on the stack abstract data type, which was implemented through the use of two functions; POP and PUSH, which manipulated the 'Stakk' data type variables. These functions were used to control the entry and exit of data onto the LIFO structure. It is important to note that this program makes use of command line arguments, and therefore must have the variable-expression input through the appropriate command line. The following source code was implemented:

```
#include<stdio.h>
#include<stdlib.h>
#include<stdbool.h>


typedef struct{ //create stack ADT
        int size;
        int max;
        float * list; //ensures exactly enough space for all integers
        } Stakk;

float POP(Stakk *stak);
bool PUSH(Stakk *stak, float entry);
void contents(Stakk *stak);
```

```
int main(int argc, char *argv[])
{
    if(argc < 4)
    {
        printf("Invalid Operation\n");
        return 1;
    }

    Stakk stak;  //declare variable of type Stakk

    stak.max = argc/2;
    stak.size = 0;
    stak.list = malloc(stak.max * sizeof(float));

    int i = 0;

    float temp1, temp2;
    temp1 = temp2 = 0.0;

    for(i=1; i<argc; i++)
    {
        if(atoi(argv[i]))
        {
            if(!PUSH(&stak,atoi(argv[i])))
            {
                printf("Unable to PUSH element on stack.\n\n");
            }
        }
        else
        {
            switch(*argv[i])
            {
            case '+':
                    temp2 = POP(&stak);
                    temp1 = POP(&stak);

                    if((temp2 == -999)||(temp1 == -999))
                    {
                        return 1;
                    }

                    PUSH(&stak, temp1+temp2);

                break;
            case '-':
                    temp2 = POP(&stak);
                    temp1 = POP(&stak);

                    if((temp2 == -999)||(temp1 == -999))
                    {
                        return 1;
                    }

                    PUSH(&stak, temp1-temp2);

                break;
            case '/':
                    temp2 = POP(&stak);
                    temp1 = POP(&stak);
                    if((temp2 == -999)||(temp1 == -999))
                    {
                        return 1;
                    }

                    PUSH(&stak, temp1/temp2);

                break;
            case 'x':
                    temp2 = POP(&stak);
                    temp1 = POP(&stak);

                    if((temp2 == -999)||(temp1 == -999))
                    {
                        return 1;
                    }

                    PUSH(&stak, temp1*temp2);

                break;
            default:
                printf("\"%s\" is an invalid operator\n\n", argv[i]);
                break;
            }
        }

        printf("\n");
        contents(&stak);
        printf("\n");
    }

    return 0;
}

float POP(Stakk *stak)
{
    if(stak->size < 1)
    {
        printf("Stack is empty.\n");
        return -999;
    }

    (stak->size)--;

    return (stak->list[stak->size]); //as size was already decremented
}

bool PUSH(Stakk *stak, float entry)
{
    if(stak->size == stak->max)
    {
        printf("Stack is full.\n");
        return false;
    }

    (stak->list[stak->size])= entry; //as stack size has not yet been
     incremented
    (stak->size)++;

    return true;
}

void contents(Stakk *stak)
{
    if(stak->size == 0)
    {
        printf("Stack is empty.\n");
        return;
    }

    int i;

    for(i=1; i<=(stak->size); i++)
    {
        printf(" |  %.2f  |\n",stak->list[(stak->size)-i]);
    }

}
```

Firstly, the command line arguments are tested, and the program is halted if less than 4 arguments are present (the executable itself is considered to be an argument). A stack data type is created through the declaration of a struct containing two integer variables which shall store the stack size and maximum size, and a float pointer, which will point to the data items stored on the ‘Stakk’. This pointer is then dynamically allocated memory through the implementation of the ‘malloc()’ function, and will later be manipulated by means of the POP and PUSH functions. An optimisation was made, whereby the stacks size was set to be that of half of the number of arguments (including the executable argument). This is because no more indexes will ever be required, and therefore memory is rationed and used efficiently.

Two functions, called the ‘POP’ and ‘PUSH’, are used to POP or PUSH data onto the stack respectively. The PUSH function simply adds an element to the first empty space in the array, while the POP function removes the last element in the array. A third function named ‘contents’ is implemented to output the contents of the stack at any moment, so as to allow the user to observe the steps involved in the RPN operation. Each function was passed the ‘stak’ address rather than the actual ‘stak’ data item, so as to reduce greatly the size of data being passed as an argument.

Each command line arguments (except the first), was tested. If the argument could be cast into an integer then it was PUSHED onto the stack, while if it couldn't, it was tested for its value through a switch statement. Each operator (+, -, x, /) was tested for, with their appropriate arithmetic operation implemented on the top two values in the stack. The result of this operation is then PUSHED back onto the top of the stack. A default clause also caught out any illegal expressions. This is then repeated until the whole expression has been evaluated.

At each stage of the operation, the contents function is called to output the contents of the stack on screen.

2.2) Testing

The programs user interface was firstly tested by inputting invalid expressions, such as illegal characters, as well as illegal expression orders. Once this test had been completed, the actual algorithm was tested for flaws and inconsistencies. This was done through multiple expression tests, where each stage of the process was observed and compared to the expected results. The final answers were also evaluated for correctness. Once this test was passed, the testing phase was over.

2.3) Screen Dumps

The following are a number of screen dumps, authenticating the positive test results obtained by this program:

```
C:\Users\Edward>main.exe 4 5
Invalid Operation

C:\Users\Edward>main.exe + 4 5
Stack is empty.
Stack is empty.

C:\Users\Edward>main.exe 5 4 + 7 - 6 x

 |  5.00  |
 |  4.00  |
 |  5.00  |


 |  9.00  |


 |  7.00  |
 |  9.00  |


 |  2.00  |


 |  6.00  |
 |  2.00  |


 |  12.00  |

C:\Users\Edward>main.exe 5 3 x 4 x 8 / 9 9 + -

 |  5.00  |


 |  3.00  |
 |  5.00  |


 |  15.00  |


 |  4.00  |
 |  15.00  |


 |  60.00  |
 |  8.00  |
 |  60.00  |


 |  7.50  |


 |  9.00  |
 |  7.50  |


 |  9.00  |
 |  9.00  |
 |  7.50  |


 |  18.00  |
 |  7.50  |


 |  -10.50  |
```

As may be seen, input validation has succeeded in halting the program when invalid data was input by the user. However, when valid data was input, the program functioned correctly, evaluating the expression through the use of the stack and the PUSH and POP functions. The contents of the stack were also output on screen with each step of the evaluation, allowing the user to observe the contents of the stack.

3) Prime Number Algorithm

3.1) Source Code

The purpose of this program is to test whether a certain number is prime (divisible only by 1 and itself) or not. It is designed to accept an integer input from the user, and return a statement explaining whether the integer is indeed prime or not. Two different approaches have been taken to this problem; the first being a traditional iterative approach, and the second implementation involving the famous Sieve of Eratosthenes. These two approached both perform the same task, however they go about this in different ways.

3.1.1) Iterative Approach

The following source code is that of the iterative approach:

```c
#include<stdio.h>
#include<stdbool.h>

bool primer(int number);

int main(void)
{
    int test;

    printf("Please enter a number:\n");
    while(scanf("%d", &test) != 1)
    {
        printf("\nInvalid entry - Please enter an integer\n");
        fflush(stdin);
    }

    if(test < 2)
    {
        printf("%d is not prime\n", test);
        return 0;
    }

    if(primer(test))
    {
        printf("%d is prime!\n", test);
    }
    else
    {
        printf("%d is not prime!\n", test);
    }
    return 0;
}

bool primer(int number)
{
    int temp, i;

    temp = (number + 1)/2; //integer division ensures that temp is half (or
    half + 0.5) of number

    for(i = 2; i <= temp; i++)
    {
        if(number % i == 0)
        {
            return false;
        }
    }

    return true;
}
```

The main program function consists mostly of user interface statements and input validation. Distinct integers such as 1, 2 and negative numbers are tested for before calling the main prime-testing function, so as to decrease the time complexity for these cases. If the entered integer does not fall under this category, then the ‘primer()’ function is called. This function accepts an integer as a parameter and iterates through from i=2 all the way to i = x/2 (where x is the integer is question), testing whether the integer is divisible by any of these numbers. There is no need to test all the way up to x-1, as if none of the integers from 2 till x/2 are divisors, then none of the others are. Also, if a divisor is in fact found, the loop is exited immediately, bypassing further wasteful tests.

If a divisor is found, then it is concluded that the integer in question is not prime. However, if the loop concludes without finding a divisor, the number is in fact prime.

3.1.2) Sieve of Eratosthenes

The following source code is that of the Sieve of Eratosthenes approach:

```
#include<stdio.h>
#define SIZE 1024

void sieve(int * arr);

int main(void)
{
    int temp;

    printf("Please enter a number between 1 and %d:\n", SIZE);

    while((scanf("%d",&temp) != 1) || (temp <1) || (temp > SIZE))
    {
        fflush(stdin);
        printf("Invalid input - Please enter an integer between 1 and
        %d\n\n",SIZE);
    }

    if(temp == 1)
    {
        printf("%d is not a prime number!\n",temp);
        return 0;
    }

    if(temp == 2)
    {
        printf("%d is indeed a prime number!\n", temp);
        return 0;
    }

    int arr[1024] = {1};
    //so as to initialise all elements except the first to 0

    sieve(arr);

    if(arr[temp-1] == 0)
    {
        printf("%d is indeed a prime number!\n", temp);
        return 0;
    }
    else
    {
        printf("%d is not a prime number!\n", temp);
    }

    return 0;
}

void sieve(int * arr)
{
    int i,j;

    for(i = 1; i < SIZE; i++)
    {
        if(arr[i] == 0)
        {
            for(j=2; j<=(SIZE/(i+1)); j++)
            //stopping condition ensures array boundaries are respected
            {
                arr[j*(i+1)-1] = 1;
        //replaces indexes which are multiples of i+1 or j with 1 (not prime)
            }
        }
    }
}
```

The first part of the program, as was previously the case, is simple user interface and data validation. However the resemblance to the previous program ends here. The next step involves declaring an array of a set size (array size may be altered through the '#define' statement). This array has its first element initialised to 1, resulting in all indexes initialised to 0 by standard. The 'sieve()' function accepts a pointer to this array, and is called to test whether the number in question is prime.

Within the sieve function, the array is manipulated so as to single out prime numbers. The array is scanned, and on the finding of an index with a 0 value, all other indexes of that multiple are set to one. This is repeated until the whole array is scanned, resulting in an array containing multiple indexes with 1 or 0 stored. Once manipulated, the array is then used to test for a specific integer. If the index of the integer in question contains a 0 then it is a prime integer, otherwise it is not.

This approach was further optimised by reducing the number of indexes in the array that where scanned to half. As was previously the case with the iterative approach, considering the first half of the integers in a list will reveal which integers in the second half are actually prime. Therefore, the scanning procedure may be stopped when half of the array has been scanned, reducing the overall scanning time-complexity by half.

3.2) Testing

3.2.1) Iterative Approach

The first step in testing this program was to run through the user validation checks and the general user interface, ensuring that data entry functions correctly. Once this was completed the functionality of the program was tested. This was done by testing random integers through the program, and comparing to results to the expected outcome. De-bugging was also used so as to trace the variable values through the iterative process. All tests concluded successfully.

3.2.2) Sieve of Eratosthenes

Once again, the user interface, including input validation, was tested with out of range or negative values, so as to ensure that data is correctly validated. The functionality of the algorithm was then put to the test, by observing the resultant array values once the ‘sieve()’ function was called. Once all appeared right, the algorithm was tested with actual integer values, with the results being compared to the expected outcome. The functionality of the optimisations made were also observed, ensuring that no problems arose.

3.3) Screen Dumps

3.3.1) Iterative Approach

The following screen dumps authenticate the functionality of this program:

```
Please enter a number:
1
Not Prime

Please enter a number:
13
13 is prime!

Please enter a number:
6625
6625 is not prime!

Please enter a number:
d

Invalid entry - Please enter an integer
33313
33313 is not prime!

Please enter a number:
-624
-624 is not prime
```

3.3.2) Sieve of Eratosthenes

The following screen dumps authenticate the functionality of the program:

```
Please enter a number between 1 and 1024:
0
Invalid input - Please enter an integer between 1 and 1024

-100
Invalid input - Please enter an integer between 1 and 1024

1030
Invalid input - Please enter an integer between 1 and 1024

d
Invalid input - Please enter an integer between 1 and 1024

7
7 is indeed a prime number!

Please enter a number between 1 and 1024:
```

```
993
993 is not a prime number!

Please enter a number between 1 and 1024:
563
563 is indeed a prime number!

Please enter a number between 1 and 1024:
1023
1023 is not a prime number!
```

4) Shell Sort Algorithm

4.1) Source code

Shell sort is a sorting algorithm used to order number in a list. The purpose of this program is to generate an array with 16384 indexes, each storing a randomly generated number, and sorting this list through the implementation of an optimised shell sort. The following is the code which was implemented:

```c
#include<stdio.h>
#include<time.h>
#include<stdlib.h>

#define SIZE 16384

void shell(int *a, int n) ;

int main(void)
{
    int * arr;
    int i;
    arr = (int *)malloc(SIZE * sizeof(int)); //dynamic memory allocation
    srand(time(NULL)); //randomiser

    for(i=0; i<SIZE; i++)
    {
        *(arr+i) = rand();
    }

    shell(arr, SIZE);

    for(i=0; i<SIZE; i++)
    {
        printf("%d) %d\n", i, *(arr+i));
    }

    return 0;
}
```

```
void shell(int * arr, int size)
{
    int gap, skim, pos, val;

    for (gap = size; gap /= 2;)
    {
        for (skim = gap; skim < size; skim++)
        {
            val = arr[skim];

            for (pos = skim; (pos >= gap) && (val < arr[pos - gap]); pos -=
            gap)
            {
                arr[pos] = arr[pos - gap];
            }

            arr[pos] = val;
        }
    }
}
```

In the above code, an array is dynamically declared with enough memory to store 16384 integers. Dynamic memory allocation is vital at this point, as it ensures that the array memory is on the heap and not on the stack. The array is then populated with random integers, through the use of the inbuilt randomiser function. The shell sort function is then called and the contents of the sorted array are output on screen.

The ‘shell()’ function accepts a pointer to an array of integers as well as an integer denoting the size of the array. Its purpose is to implement shell sorting upon the array. This involves the generation of virtual gaps in the array. The gap starts of as the total size of the array, and with each iteration reduces in size by half. Elements at a certain index in the array are compared to elements outside of their gap, and are swapped if the correct order is disturbed.

With each iteration, the gap decreases. This is an optimisation of the traditional shell sort, as at first, indexes are allowed to be placed into a broader, more general position due to the large gap; however as the gap decreases, the elements are moved into their final exact position. This optimisation often reduces the time complexity for sorting from O(n^2) to O(n.log(n)).

4.2) Testing

No user interaction is present in this program, as the array is declared, populated and sorted without the need of user input. Therefore no testing was required for data validation. The first tests on the shell sort involved smaller arrays, to ensure that the functionality of the sorting algorithm was fine. This was then scaled up to the full array and tested multiple times, with randomised values of different scales (from 1-10 all the way to 1- INT_MAX). These tests ensured that the program was functioning properly and that no bugs or pitfalls were present.

4.3) Screen Dumps

The following screen dumps authenticate the functionality of the above code:

For the first test, the array size was scaled down to only ten elements, for observation purposes:

```
1) 1812
2) 7869
3) 9771
4) 14561
5) 20458
6) 21898
7) 22196
8) 22820
9) 30505
10) 31685
```

Scaling up to 50 elements:

```
1) 780
2) 1724
3) 3153
4) 3272
5) 4947
6) 6086
7) 7861
8) 7961
9) 8133
10) 8250
```

```
11) 8540
12) 8697
13) 9995
14) 10215
15) 11781
16) 12551
17) 12656
18) 13758
19) 13860
20) 13985
21) 14761
22) 15087
23) 15097
24) 15265
25) 16648
26) 17089
27) 18556
28) 18988
29) 19519
30) 20313
31) 21083
32) 22070
33) 23963
34) 24456
35) 24545
36) 26581
37) 26995
38) 27402
39) 28424
40) 28432
41) 28697
42) 29013
43) 30841
44) 30925
45) 31009
46) 31383
47) 31734
48) 32348
49) 32483
50) 32597
```

Finally, the full 16384 elements (due to the vast number of elements, only a fraction are visible on the command window):

```
16311) 32637
16312) 32637
16313) 32638
16314) 32638
16315) 32639
16316) 32640
16317) 32642
16318) 32643
16319) 32644
```

```
16320) 32645
16321) 32645
16322) 32645
16323) 32651
16324) 32654
16325) 32659
16326) 32661
16327) 32664
16328) 32673
16329) 32677
16330) 32677
16331) 32677
16332) 32681
16333) 32682
16334) 32683
16335) 32683
16336) 32687
16337) 32687
16338) 32692
16339) 32693
16340) 32694
16341) 32695
16342) 32695
16343) 32700
16344) 32702
16345) 32704
16346) 32704
16347) 32706
16348) 32708
16349) 32710
16350) 32710
16351) 32712
16352) 32712
16353) 32716
16354) 32716
16355) 32718
16356) 32719
16357) 32723
16358) 32723
16359) 32724
16360) 32726
16361) 32727
16362) 32729
16363) 32730
16364) 32731
16365) 32732
16366) 32735
16367) 32739
16368) 32740
16369) 32745
16370) 32746
16371) 32746
16372) 32747
16373) 32751
16374) 32752
16375) 32752
16376) 32752
16377) 32755
16378) 32757
```

```
16379) 32760
16380) 32761
16381) 32761
16382) 32762
16383) 32765
16384) 32766
```

As may be seen, all the elements output on screen are in order. This proves that the optimised shell sort algorithm does indeed work, and is able to sort arrays of a vast size such as the one observed above.

5) Square Root Algorithm

5.1) Source Code

This algorithm is designed to calculate an estimate for the square root of a given integer/decimal. The iterative method that is the Babylonian method for estimating the square root of a number has been chosen to be implemented. This method involves the calculation of a seed value, which will be used as a starting point for the computation. This seed, together with the original number, is inputted into an equation several times, each time yielding a better estimate for the square root. The following code was implemented in this program.

```c
#include<stdio.h>
#include<stdlib.h>
#include<math.h>

#define ACCURACY 0.0001
#define SEED 1

float rooter(float num);
int seeder(float num, int n);

int main(void)
{
    float num, answer;

    printf("Please enter a positive number whose square root you would like
    to find:\n");

    while((scanf("%f", &num) != 1) || (num <=0))
    {
        fflush(stdin);
        printf("Invalid input. Please enter a positive integer/float.\n\n");
    }

    answer = rooter(num);
    printf("The square root of %.4f is: %.6f\n\n",num,answer);

    return 0;
}

float rooter(float num)
{
    float seed, answer, diff;
    answer = 0.0;
    int tseed;

    tseed = seeder(num,SEED);

    if(tseed % 2 == 0)
    //mathematical manner of finding rough estimate for square root
    {
        seed = 6*(pow(10,((tseed-2)/2)));
    }
    else
    {
        seed = 2*(pow(10,((tseed-1)/2)));
    }

    while((diff>ACCURACY)||(diff<-ACCURACY))
    {
        answer = (0.5*(seed + (num/seed)));
        diff = seed-answer;
        seed = answer;
    }

    return answer;
}

int seeder(float num, int n)
{
    while((num/=10)>10)
    {
        n++;
    }

    return ++n;
}
```

The user is asked to enter the number whose square is to be found. Data validation is enforced to ensure that the input is a positive float or integer. Following validation, the ‘rooter()’ function is called and is passed the initial value as a parameter. This function in turn calls the ‘seeder()’ function, which is used to generate the appropriate seed value. This value is based on the number of digits in the initial data.

The seed value is then returned to the ‘rooter()’ function, where the final seed value is calculated, based on further equations of the Babylonian method. Finally, the calculated seed value is inputted into an equation together with the initial value, so as to compute an estimate for the square root. The increase in accuracy is calculated and the process is repeated until the change in value between each cycle is less than that of the specified accuracy (set to 0.0001). The final root value is then output on screen.

5.2) Testing

The first components to be tested were that of the user interface and data validation. Incorrect and negative values were input so as to test the integrity of the program. Next, the seeder function was tested to ensure that the correct seed value is generated each time. This is vital as suitable seeds could reduce the number of required iterations greatly. Different tester values where input and, through the use of a de-bugger, variables where traced to ensure that functionality is maintained throughout.

Once the ‘seeder()’ was tested, the ‘rooter()’ was also tested. Different values where input, and the respective answer was compared to the true answer by means of a calculator. Slight adjustments were made to the equation until a high degree of accuracy was maintained.

5.3) Screen Dumps

The following screen dumps authenticate the functionality of the above code:

```
Please enter a positive number whose square root you would like to find:
-10
Invalid input. Please enter a positive integer/float.

d
Invalid input. Please enter a positive integer/float.

24
The square root of 24.0000 is: 4.898980

Please enter a positive number whose square root you would like to find:
3764
```

```
The square root of 3764.0000 is: 61.351448

Please enter a positive number whose square root you would like to find:
912.74
The square root of 912.7400 is: 30.211588

Please enter a positive number whose square root you would like to find:
817257.2
The square root of 817257.2000 is: 904.022766
```

As may be seen, data validation measures ensured that invalid data did not halt the program, but instead prompted the user to re-enter a legal value. Also, one may observe that both integer values, as well as decimal values were handled well and the results were of a high degree of accuracy, matching calculator results up to 4 decimal places.

6) Matrix Multiplication Algorithm

6.1) Source Code

The purpose of this program is to generate two 16 X 16 matrices, populate them with random values, and perform matrix multiplication, storing the values in a new variable. The following code was implemented:

```
#include<stdio.h>
#include<stdlib.h>
#include<time.h>

#define WIDTH 16
#define HEIGHT 16
#define BASE 10

void matrix(double mat1[WIDTH][HEIGHT], double mat2[WIDTH][HEIGHT], double
temp[WIDTH][HEIGHT]);

int main(void)
{
    int i,j,k;
    double mat1[WIDTH][HEIGHT], mat2[WIDTH][HEIGHT], temp[WIDTH][HEIGHT];
    srand(time(NULL));
    for(i = 0; i <WIDTH; i++)
    {
        for(j = 0; j <HEIGHT; j++)
        {
            mat1[i][j] = (rand()%BASE) + 1;
            //populated matrices with random numbers between 1 and 10
            mat2[i][j] = (rand()%BASE) + 1;
            temp[i][j] = 0;
        }
    }

    for(i=0; i < HEIGHT; i++)
    {
        for(j=0; j < WIDTH; j++)
        {
            for(k=0; k < WIDTH; k++)
            {
                temp[j][i] += (mat1[k][i])*(mat2[j][k]);
            }
        }
    }

    for(i = 0; i <WIDTH; i++)
    {
        for(j = 0; j <HEIGHT; j++)
        {
          printf(" %.2f ", temp[i][j]);
        }
        printf("\n\n");
    }

    return 0;
}
```

At the start of the program, three two-dimensional arrays of type integer are declared, each of dimensions 16 X 16. These first two matrices named ‘mat1’ and ‘mat2’ are then populated with random numbers by means of the randomiser function, while the third, ‘temp’, is left void to be later populated.

Matrix multiplication then commences by means of an iterative approach, whereby each individual element on a row on the first matrix is multiplied by each individual element on a column on the second matrix. The values are then summed and stored in the corresponding index of the array ‘temp’.

Finally, the ‘temp’ matrix is skimmed through, having each of its indexes output on screen in the visual order that they occur.

6.2) Testing

Requiring no user input, this program had no input validation in place and therefore none to test. However, extensive testing was required upon the matrix multiplication until the process was successful. For testing purposes, the dimensions of the matrixes were reduced greatly, and values where manually input into the matrices, so as to observe the procedure. Once the smaller scale tests proved successful, the dimensions where increased to that of 16 X 16. Finally, the randomiser function was implemented, populating the arrays with random values.

6.3) Screen Dumps

The following screen dumps authenticate the functionality of the above program:

Setting dimensions to 3 X 3 and manually inputting ascending values of the following type:

```
1.00   2.00   3.00
2.00   3.00   4.00
3.00   4.00   5.00
```

Result:

```
14.00   20.00   26.00

20.00   29.00   38.00

26.00   38.00   50.00
```

Setting dimensions to 3 X 3 and implementing the randomiser function:

```
 174.00  107.00  55.00

  48.00  31.00  18.00

  82.00  55.00  27.00
```

Setting dimensions to 16 X 16 and implementing the randomiser function:

```
423.00   397.00   341.00   405.00   366.00   491.00   481.00   392.00   511.00
490.00
 320.00  367.00  452.00  385.00  374.00  477.00
```

```
 560.00   553.00   447.00   494.00   504.00   632.00   577.00   491.00   691.00
687.00
 488.00  408.00  589.00  449.00  475.00  636.00
```

```
 614.00   665.00   502.00   524.00   603.00   689.00   578.00   539.00   671.00
690.00
 526.00  548.00  660.00  548.00  513.00  705.00
```

```
 636.00   571.00   440.00   591.00   590.00   672.00   613.00   567.00   766.00
682.00
 500.00  528.00  636.00  583.00  540.00  743.00
```

```
 571.00   485.00   428.00   411.00   485.00   487.00   525.00   423.00   551.00
599.00
 424.00  418.00  484.00  355.00  427.00  582.00
```

```
 625.00   602.00   434.00   582.00   577.00   663.00   642.00   557.00   719.00
661.00
 488.00  543.00  587.00  581.00  493.00  731.00
```

```
 507.00   528.00   392.00   385.00   461.00   488.00   428.00   369.00   499.00
575.00
 431.00  350.00  476.00  372.00  335.00  479.00
```

```
 468.00   481.00   378.00   409.00   434.00   570.00   484.00   394.00   556.00
563.00
 454.00  395.00  549.00  427.00  439.00  556.00
```

```
 572.00   521.00   383.00   504.00   463.00   588.00   511.00   479.00   617.00
606.00
 492.00  386.00  502.00  432.00  411.00  599.00
```

```
455.00   426.00   386.00   380.00   448.00   508.00   542.00   426.00   537.00
553.00
 411.00  440.00  474.00  375.00  448.00  522.00
```

```
 496.00   482.00   376.00   467.00   396.00   540.00   485.00   419.00   506.00
523.00
 388.00  413.00  432.00  417.00  385.00  504.00
```

```
 512.00   487.00   393.00   454.00   487.00   596.00   513.00   448.00   573.00
609.00
 434.00  445.00  524.00  519.00  459.00  596.00
```

```
 454.00   401.00   347.00   400.00   396.00   477.00   489.00   368.00   519.00
523.00
 363.00   358.00   454.00   417.00   398.00   521.00
________________________________________________________________________________

 557.00   574.00   436.00   427.00   538.00   612.00   537.00   432.00   606.00
686.00
 488.00   429.00   602.00   503.00   429.00   599.00
________________________________________________________________________________

 629.00   523.00   410.00   534.00   481.00   574.00   558.00   441.00   584.00
583.00
 465.00   477.00   488.00   422.00   444.00   602.00
________________________________________________________________________________

 429.00   428.00   366.00   350.00   378.00   423.00   435.00   365.00   465.00
446.00
 325.00   372.00   453.00   343.00   369.00   488.00
________________________________________________________________________________
```

N.B Due to large dimensions row values stretch across multiple lines

As may be seen above, the algorithm functions with small dimensions, as well as larger dimensions. The first screen dump was the result of manually entering values into the program, so as to demonstrate the success of algorithm, while all other screen dumps were the results of random valued matrices being multiplied.

7) Maximum Integer Search Program

7.1) Source Code

The purpose of this program is to recursively search for the largest number in a given list of variable length. The following source code was implemented in the above program:

```
#include<stdio.h>
#include<limits.h>

static int size;
static float m = INT_MIN;
static int i;

void max(float []);

int main(void)
{
    int i;

    printf("Please enter the size of your list: ");
    while((scanf("%d",&size)!=1)||(size<1))
    {
        fflush(stdin);
        printf("\nInvalid input - Please enter a positive integer.\n\n");
    }

    float arr[size];

    printf("Please enter %d integers:\n", size);
    for(i=0; i<size; i++)
    {
        printf("\n%d) ",i+1);

        while(scanf("%f",&arr[i])!= 1)
        {
            fflush(stdin);
            printf("Invalid Input!\n%d",i+1);
        }
    }

    max(arr);

    printf("The greatest number in the list is %.2f\n\n", m);

    return 0;
}


void max(float arr[size])
{
    while(i<size) //i initialised to 0
    {
        if(arr[i] > m)
        {
            m = arr[i];
        }

        i++;

        max(arr);
    }
}
```

At the start of the program the user is asked to specify the size of the array to be declared. Once input is validated, the array of type float is declared and then populated by the user. The 'max()' function is then called, initiating the procedure to find the largest number in the list.

This function operates in a recursive manner, comparing the value in each index to a benchmark maximum value (initially set to the greatest negative int number). If the number in the current index is greater than the maximum value then the maximum value is set to the current number. A recursive call is then made to the same function, passing the same array as a parameter.

With each recursive call, the next index is tested, until finally the whole list has been tested. This recursive call is only possible due to the static variables ‘size’, ‘i’ and ‘max’. This is because with each recursive call, these variables remain unchanged, thus keeping track of the next index to visit, as well as the current max and the size of the array.

Once all recursive levels have been exited, the maximum number is output on screen.

7.2) Testing

The first task was to test the user interface and data validation for user input. This was accomplished by entering a range of different integer values as well as characters, to ensure that no invalid input halted program execution.

Next, the ‘max()’ function was tested for functionality by submitting arrays of different sizes and containing different values. Smaller arrays were initially submitted, with whole integer values. However, once these submissions passed the test, larger array containing float values where entered. All tests where passed with the correct value being submitted as the max.

7.3) Screen Dumps

The following screen dumps authenticate the functionality of the program discussed above:

```
Please enter the size of your list: -3

Invalid input - Please enter a positive integer.

0

Invalid input - Please enter a positive integer.
```

```
d

Invalid input - Please enter a positive integer.

4
Please enter 4 integers:

1) d
Invalid Input!
1)12.2

2) 34.1

3) -98.3

4) 52
The greatest number in the list is 52.00

Please enter the size of your list: 10
Please enter 10 integers:

1) 84.2

2) 32.1

3) -12

4) 102.22

5) 98.2

6) 75.6

7) 23

8) 74

9) 0.12

10) 9.7
The greatest number in the list is 102.22
```

As may be seen, invalid data entry was correctly dealt with, with the user being prompted to re-enter a valid value when it came to specifying the array size as well as when populating the array. Also one may observe that all array sizes and values where handled properly, with the largest number always being displayed on screen.

8) Trigonometric Calculator Algorithm

8.1) Source Code

The purpose of this program is to compute sine and cosine values for angles submitted by the user, up to a certain degree of accuracy. This was achieved through the use of Taylor's series for sine and cosine. The following source code was implemented in this program:

```
#include<stdio.h>
#include<math.h>

float expansion_cos(float x, int terms);
float expansion_sin(float x, int terms);
long long factorial(int num);

int main(void)
{
    float x, answer;
    int terms, choice;

    printf("Please select the desired trigonometric function:\n");
    printf("1) cos(x)                2) sin(x)\n\n");

    while((scanf("%d",&choice) != 1) || (choice < 1) || (choice > 2))
    {
        fflush(stdin);
        printf("Incorrect input\n");
    }

    printf("Please enter the radian cosine angle: \n");
    while(scanf("%f", &x)!=1)
    {
        fflush(stdin);
        printf("\nInvalid input - Please enter a radian angle.\n\n");
    }

    printf("\nPlease enter the number of terms to which you would like to
    calculate: \n");

    while((scanf("%d", &terms) != 1) || (terms < 2))
    {
        fflush(stdin);
        printf("\nInvalid input - Please enter a positive integer greater
        than 1.\n\n");
    }

    if(terms>17)
    {
        terms = 17; //as negligible difference with excess terms
    }

    switch(choice)
    {
    case 1:
        answer = expansion_cos(x, terms);
        break;
    case 2:
        answer = expansion_sin(x,terms);
        break;
    default:
        printf("Error\n");
        break;
    }

    printf("\nThe series expansion of %s(%.2f) to %d terms is: %.3f\n",
    choice == 1? "cos":"sin",x,terms,answer);

    return 0;
}

Long long factorial(int num)
{
    int i;
    long long ans = 1;

    for(i=num; i>1; i--)
    {
        ans*=i;
    }

    return ans;
}

float expansion_cos(float x, int terms)
{
    int i;
    float value = 0.0;

    for(i=0; i<terms; i++)
    {
        value += (pow(-1,i))*(pow(x,2*i))/factorial(2*i);
    }

    return value;
}

float expansion_sin(float x, int terms)
{
    int i;
    float value = 0.0;

    for(i=1; i<=terms; i++)
    {
        value += (pow(-1,i-1))*(pow(x,(2*i)-1))/factorial((2*i)-1);
    }

    return value;
}
```

As may be seen above, the first step in this program is for the user to select between the cosine or sine function calculator. The user is then asked to enter a radian angle to calculate the appropriate value of. Finally the user is prompted to specify the number of terms to be considered when calculating the trigonometric value using Taylor's theorem.

The appropriate sine or cosine function is then called according to the user's initial selection. The radian angle together with the accuracy in number of terms are passed as parameters.

Inside the appropriate sine and cosine functions, the series expansion is calculated through the use of a loop. The 'value' variable accumulates with each iteration of the loop, each time getting closer to the final value. Within this equation, factorial values are required; therefore a factorial function is called to calculate the respective factorial of a given number. This function operates in an iterative manner, accumulating the factorial value.

These functions work together to calculate an accurate value for these trigonometric functions. The final value is then output on screen.

8.2) Testing

As usual, the first set of tests include input validation testing so as to ensure that no user input could halt the program. Following this test, the factorial function was tested to ensure that it was running properly. A number of different values were entered and, together with the tracer, the variable values as well as the final output where observed. During this test, it was noted that due to the accumulation of extremely large values, the data type of the factorial value had to be of type 'long long' so as to accommodate for these vast numbers. Furthermore, a limit of 17 terms had to be set on the accuracy of Taylor's series so as not to overflow the variable storing the factorial value. This ensures a high level of accuracy while maintaining data integrity.

Different radian angles where then input into these functions, and the observed output was compared to the expected result by means of a calculator. Once these functions were observed to produce highly accurate answers, they were considered fully functional.

8.3) Screen Dumps

The following screen dumps serve to authenticate the functionality of the above mentioned algorithm:

```
Please select the desired trigonometric function:
1) cos(x)                2) sin(x)

3
Incorrect input
d
Incorrect input
1
Please enter the radian cosine angle:
3.14159

Please enter the number of terms to which you would like to calculate:
10

The series expansion of cos(3.14) to 10 terms is: -1.000

Please select the desired trigonometric function:
1) cos(x)                2) sin(x)

2
Please enter the radian cosine angle:
2.563

Please enter the number of terms to which you would like to calculate:
15

The series expansion of sin(2.56) to 15 terms is: 0.547

Please select the desired trigonometric function:
1) cos(x)                2) sin(x)

1
Please enter the radian cosine angle:
-0.734

Please enter the number of terms to which you would like to calculate:
20

The series expansion of cos(-0.73) to 17 terms is: 0.743
```

As may be observed above, the algorithm is functional and operates to a certain degree of accuracy, with almost all computations with at least ten terms yielding an accuracy up to four decimal places. Also, another noteworthy feature is that in the last case the number of terms was restricted to 17 so as to prevent an overflow within the factorial function.

9) Palindrome Algorithm

9.1) Source Code

The purpose of this program is to detect whether an input string is a palindrome (symmetrical) or not. The following source code was implemented in this program:

```
#include<stdio.h>
#include<string.h>
#include<ctype.h>

#define SIZE 25

int main(void)
{
    char str[SIZE];
    char temp[SIZE];
    int size, i;

    printf("Please input a string:\n");
    while(scanf("%s",str) != 1)
    {
        fflush(stdin);
        printf("Invalid Input - Please input a string");
    }

    size = strlen(str);

    if(size == 1)
    {
        printf("%s is not a palindrome!\n", str);
        return 0;
    }

    strlwr(str);

    for(i=0; i<size; i++)
    {
        temp[i] = str[size -(i+1)];
    }

    temp[size] = '\0'; //so as to disregard the \n
    if (strcmp(temp, str) == 0)
    {
        printf("%s is infact a palindrome!", str);
        return 0;
    }

    printf("%s is not a palindrome!", str);
    return 0;
}
```

To test a string, a string is obviously required. Therefore the user is prompted to enter a string to be tested. The size of the string is then computed and stored for later use. By means of the ctype function 'strlwr()' the string is converted to lower case so as to make the palindrome test case insensitive. The string is then iteratively reversed, storing the reversed version in a different character array. The two character arrays are then compared by means of the 'strcmp' function, which will reveal whether the input string is in fact a palindrome.

If the input string is only one character, the string is definitely not a palindrome, therefore an optimisation was made to terminate execution early in the case of a one-character string.

9.2) Testing

Input validation was initially tested to ensure invalid data doesn't halt the program execution. Further tests on the string reversal feature were carried out by inputting random string sequences and observing the output. These tests highlighted a key problem with the '\n' entry at the end of the character string. This escape character was also being reversed, and therefore required to be overridden by a '\o' so as to allow palindrome comparison.

Finally, the functionality of the algorithm was tested by submitting a number of different strings, some palindromes and other not, and comparing the observed output to the expected output.

9.3) Screen Dumps

The following screen dumps authenticate the functionality of the above mentioned algorithm:

```
Please input a string:
hello
hello is not a palindrome!

Please input a string:
HaNnah
hannah is infact a palindrome!

Please input a string:
hannnah
hannnah is infact a palindrome!

Please input a string:
12121
12121 is infact a palindrome!

Please input a string:
12312
12312 is not a palindrome!

Please input a string:
l
l is not a palindrome!
```

As may be observed, the algorithm is fully functional and is able to decipher between a string which is a palindrome, and one which is not. Also, comparison was case insensitive, as may be observed in the second test.

Statement of Completion

I, Edward Fleri Soler, state that I have attempted all problems, personally testing each problem, each one being fully functional and producing satisfactory/expected results.