CSA1017 - Assignment

Daniel Magro
484497M

May 2016

Statement of Completion

The questions below are the ones that have been attempted:

Question 1 This question has been successfully completed.

Question 2 This question has been successfully completed.

Question 3 This question has been successfully completed.

Question 4 This question has been successfully completed.

Question 5 This question has been successfully completed.

Question 6 This question has been successfully completed.

Question 7 This question has been successfully completed.

Question 8 This question has been successfully completed.

Question 9 This question has been successfully completed.

Signature

Date

Contents

Question 1	**3**
Question 2	**6**
Question 3	**11**
Question 4	**15**
Question 5	**19**
Question 6	**22**
Question 7	**26**
Question 8	**29**
Question 9	**33**

Question 1

This problem was completed successfully and works as intended.

```c
#include <stdio.h>
#include <stdlib.h> // for exit function
#include <string.h> // for strcat function

void convertRoman(int number);

int main() {
  setbuf(stdout, NULL);

  // variable which will store the number entered by the user that is to be represented in Roman Numerals
  int number;
  // prompt the user to enter a number between 1 and 1024 to be represented in Roman Numerals
  printf("Please enter the number (1-1024) you wish to be represented in Roman Numerals\n");
  // Accept the input from the user
  scanf("%d", &number);
  // Check if the user's input is greater than 1 and less than 1024
  if (number >= 1 && number <= 1024) {
    // if so call the method which will represent it as a roman numeral
    convertRoman(number);
  } else {
    // if not notify the user and exit the program
    printf("Only numbers between 1 and 1024 are accepted as input.\n");
    exit(0);
  }
  return 0;
}

```

```c
void convertRoman(int number) {

  printf("The number %d will be represented in Roman Numerals.\n", number);

  // an array of the symbols of the numbers used in the Roman Numerals system
  char romanNumerals[13][3] = {"M", "CM", "D", "CD", "C", "XC", "L", "XL", "X", "IX", "V", "IV", "I"};
  // an array of the values of the numbers used in the Roman Numerals system
  int numerals[13] = {1000, 900, 500, 400, 100, 90, 50, 40, 10, 9, 5, 4, 1};

  // array of chars that holds the resulting Roman Numeral
  char roman[15];
  roman[0] = '\0';
  // variable which indicates which Roman Numeral is currently being considered
  int i = 0;
  // variable which holds how many more numbers need to be converted
  int r = number;

  // while there are still numbers to be converted to Roman Numerals
  while(r != 0){
    // if the Roman Numeral currently being considered were to be deducted from the remaining value, and the value would
    // remain greater than or equal to zero, than the calculation is carried out, and the Roman Numeral is stored in
    // the resulting array
    if( (r-numerals[i]) >= 0){
      r -= numerals[i];
      strcat(roman, romanNumerals[i]);
    }
    // else consider the next Roman Numeral in the array
    else{
      i++;
    }
  }

  // print the result
  printf("The inputted number in Roman Numerals is: %s\n", roman);
}
```

Table 1.1: Testing: Question 1

Input	**Expected Output**	**Actual Output**
0	Input not accepted	Input not accepted
1	I	I
72	LXXII	LXXII
652	DCLII	DCLII
999	CMXCIX	CMXCIX
1024	MXXIV	MXXIV
1100	Input not accepted	Input not accepted

Question 2

This problem was completed successfully and works as intended.

```c
#include <stdio.h>
#include <stdlib.h> // for atof function
#include <string.h> // for strlen function
#include <ctype.h>  // for isdigit function

#define SIZE 100  // size of the array which will be used as the stack

void push(float stack[], int * stackPointer, float si); // si is stack input
float pop(float stack[], int * stackPointer);

int main() {
  setbuf(stdout, NULL);

  // declaring the array which will be used as a stack
  // and a pointer to the stack (stackPointer) which points to the last character pushed onto the stack.
  float stack[SIZE];
  int stackPointer = -1;

  // declaring an array of 'char's which will hold the postfix expression to be evaluated
  char postfix[SIZE];
  // ask the user to input the expression to be converted
  printf("Enter the RPN expression you would like evaluated (include a space between operands and operators) \n");
  // store the user's inputted expression in the array 'expression'
  gets(postfix);

  // Evaluating postfix expressions using the stack
  int i = 0;
```

```c
float op1;        // float that will hold one operand popped from the stack
float op2;        // float that will hold one operand popped from the stack
float result;     // float that will temporarily hold the result of an operation
char stringOperand[10]; // array will be used to convert strings to numbers
int operandIndex;   // index for the array of the operand as a char
float floatOperand;   // float that will hold converted strings

while(postfix[i] != '\0'){

  // if the current character is an operand
  if(isdigit(postfix[i])){
    // reset the index for the array containing the operand as a string to -1
    operandIndex = -1;
    // append the current character (digit) to the array storing the operand as a string
    stringOperand[++operandIndex] = postfix[i];
    // while the next character isn't a whitespace (implies it is part of the same number)
    while(postfix[i+1] != ' '){
      stringOperand[++operandIndex] = postfix[++i];
    }
    // indicate the end of the string
    stringOperand[++operandIndex] = '\0';

    // convert the string containing the operand to a float
    floatOperand = atof(stringOperand);
    // push the float to the stack
    push(stack, &stackPointer, floatOperand);
  }
  // if the current character is an operator
  else if(postfix[i] == '+' || postfix[i] == '-' || postfix[i] == '*' || postfix[i] == '/'){
    switch (postfix[i]){

    // if the operator is a + (addition)
    case '+':
      // set the second operand to whatever is at the top of the stack
      op2 = pop(stack, &stackPointer);
      // set the first operand to whatever is at the top of the stack
      op1 = pop(stack, &stackPointer);
```

```
        // compute the result
        result = op1 + op2;
        // push the result to the top of the stack
        push(stack, &stackPointer, result);
        break;
      case '-':
        op2 = pop(stack, &stackPointer);
        op1 = pop(stack, &stackPointer);
        result = op1 - op2;
        push(stack, &stackPointer, result);
        break;
      case '*':
        op2 = pop(stack, &stackPointer);
        op1 = pop(stack, &stackPointer);
        result = op1 * op2;
        push(stack, &stackPointer, result);
        break;
      case '/':
        op2 = pop(stack, &stackPointer);
        op1 = pop(stack, &stackPointer);
        result = op1 / op2;
        push(stack, &stackPointer, result);
        break;
      }
    }
    i++;
  }

  printf("The result of the expression (rounded to 4 decimal places) is %.4f\n", pop(stack, &stackPointer));

  return 0;
}

// method to push onto the stack
void push(float stack[], int * stackPointer, float si) {
  // if the stack pointer points to the last element, then the stack is full
  if (*stackPointer == (SIZE - 1)) {
    printf("The stack is full\n");
  } else {
    (*stackPointer)++;      // increment the stack pointer to point to an empty element
    stack[*stackPointer] = si;  // stores the input in the element indicated by the stack pointer
  }

  // Displaying the contents of the stack, the element which is displayed first is the one at the bottom of the stack
```

```
  int i = 0;
  while(i <= *stackPointer){
    printf("%f", stack[i]);
    printf("\t");
    i++;
  }
  printf("\n");
}

// method to pop from the top of the stack
float pop(float stack[], int * stackPointer) {
  // if the stack pointer points to element '-1', then the stack is empty
  if (*stackPointer == -1) {
    printf("The stack is empty\n");
    return '\0';  // return a null character
  } else {
    // return whatever is at the top of the stack (pointed to by the stack pointer)
    // then decrement the stack pointer to point to the previous element (accomplished with a postfix decrement)
    return stack[(*stackPointer)--];
  }
}
```

Table 2.1: Testing: Question 2

Input	Expected Output	Actual Output
25 32 +	57.0000	57.0000
67 99 * 32 + 66 - 99 88 + /	35.2888	35.2888
3098.6543 988.266 32.23 - 52 * 65.55 - *	153842976.0000	153842976.0000

Question 3

This problem was completed successfully and works as intended.

```c
#include <stdio.h>
#include <stdbool.h>  // for boolean data type
#include <math.h>   // for sqrt function

bool is_prime(int number);
int sieve_of_Eratosthenes(int limit);

int main() {
  setbuf(stdout, NULL);

  int number;
  printf("Which number do you want to know is prime?\n");
  scanf("%d", &number);

  if(is_prime(number) == 1){
    printf("%d is a prime number\n", number);
  } else{
    printf("%d is not a prime number\n", number);
  }

  printf("Up till which number would you like to know is
   prime?\n");
  printf("Using the Sieve of Eratosthenes algorithm.\n");
  int limit;
  scanf("%d", &limit);
  sieve_of_Eratosthenes(limit);

  return 0;
}

bool is_prime(int number) {

  // if the number is 1, then it is not a prime number
```

```
  if(number == 1){
    return false;
  }
  // else, if the number is 2, then it is a prime number
  else if(number == 2){
    return true;
  }
  // else, if the number is even, then it is not a prime number
  else if(number%2 == 0){
    return false;
  }
  // else, check if the number is divisible by any odd number between 3 and the square root of the number
  else{
    int i;
    for(i=3; i<=sqrt(number); i+=2){
      if(number%i == 0){
        return false;
      }
    }
    return true;
  }
}

int sieve_of_Eratosthenes(int limit){

  // a boolean array which holds whether the corresponding number is a prime or not
  // for simplicity the array will store from 0-limit
  bool primes[limit+1];

  // setting the number one as a non-prime
  primes[1] = false;
  // setting all the elements of the array of primes (2 - limit) to true
  int i,j;  // counters used for loops
  for(i=2; i<(limit+1); i++){
    primes[i] = true;
  }

  // go through all the numbers
  for(i=2; i<(limit+1); i++){
    // if the number is so far considered a prime
    if(primes[i] == 1){
      // start from twice the current number (since it is surely not a prime), and mark it and every successive
      // (with the number itself as an interval) number
```

```
      // Eg: If we are on number 5, then start from (2*5) 10
    and mark it as non-prime, as well as every other 5th
    number.
      for(j=2*i; j<(limit+1); j+=i){
        primes[j] = false;
      }
    }
  }

  // printing the prime numbers
  for(i=1; i<(limit+1); i++){
    if(primes[i] == 1){
      printf("%d\t", i);
    }
  }
  printf("\n");

  return 0;
}
```

Table 3.1: Testing: Question 3

Input	Expected Output	Actual Output
1, 10	Not Prime, 2 3 5 7	Not Prime, 2 3 5 7
2, 2	Prime, 2	Prime, 2
777743, 1	Prime,	Prime,
777757, 5	Not Prime, 2 3 5	Not Prime, 2 3 5
11111111, 7	Not Prime, 2 3 5 7	Not Prime, 2 3 5 7
11111117, 9	Prime, 2 3 5 7	Prime, 2 3 5 7

Question 4

This problem was completed successfully and works as intended.

```c
#include <stdio.h>
#include <stdlib.h>   // for exit function
#include <time.h>
#include <stdbool.h>  // for boolean data type
#include <math.h>   // for ceil function

#define SIZE 16384  // Setting the size of the array to 16384
#define RANGE 99999 // Range of random numbers (1 - range)

void swap(int * u, int * v);  // Generic swap method
void shellSort(int array[]);

int main(){
  setbuf(stdout, NULL);

  // declaring an array of 16384 integers
  int array[SIZE];

  // filling the array with random number
  srand(time(NULL));  // The seed of the random numbers will be based on time
  int i;  // counter for loops
  for(i=0; i<SIZE; i++){
    array[i] = (rand() % RANGE) + 1;
  }

  // displaying the unsorted list
  for(i=0; i<SIZE; i++){
    printf("%d ", array[i]);
  }
  printf("\n");

  // calling the optimised shell sort method
```

```
  shellSort(array);

  // displaying the sorted list
  for(i=0; i<SIZE; i++){
    printf("%d ", array[i]);
  }
  printf("\n");

  // Testing if the array was successfully sorted
  for (i = 1; i < SIZE; i++) {
    // if the preceding element is larger than the next one, then the program will output that the list wasn't
    // successfully sorted and exit the program
    if (array[i - 1] > array[i]) {  // each element is compared to the one before it
      printf("The list was not successfully sorted\n");
      exit(1);
    }
  }
  printf("The list was successfully sorted\n");

  return 0;
}

void shellSort(int array[]){
  // int which holds the gap which will be used to sort smaller 'sublists'
  int gap;
  // int which will be used as counters for loops
  int i;
  // boolean value which will indicate whether a swap has occurred or not
  bool flag;

  // setting the value of the gap/interval to half of the size of the array
  gap = SIZE/2;

  // do the following, until no swaps have occurred and the gap has become 1
  do{
    // set the flag to false
    flag = false;
    // bubble sort, but instead of comparing adjacent elements, elements have the gap in between
    for(i=0; i<(SIZE-gap); i++){
      if(array[i] > array[i+gap]){
        swap(&array[i], &array[i+gap]);
```

```c
        // set the flag to true to indicate that a swap has occurred
        flag = true;
      }
    }
    // if the gap is still greater than 1
    if(gap > 1){
      // then divide it by 2 and round up
      gap = ceil(gap/2);
      // set the flag to true to indicate that the gap is not yet 1
      flag = true;
    }
  }while(flag != false);
}

// Generic swap method
void swap(int *u, int *v) {
  int temp;

  temp = *u;
  *u = *v;
  *v = temp;
}
```

Testing: Question 4

This Problem was tested for correctness by using the following code:

```c
// Testing if the array was successfully sorted
for (i = 1; i < SIZE; i++) {
  // if the preceding element is larger than the next one, then the program will output that the list wasn't
  // successfully sorted and exit the program
  if (array[i - 1] > array[i]) {  // each element is compared to the one before it
    printf("The list was not successfully sorted\n");
    exit(1);
  }
}
printf("The list was successfully sorted\n");
```

Question 5

This problem was completed successfully and works as intended.
The following image was used as a guide to model the program to the Newton-Raphson Algorithm.

$$
\begin{aligned}
x_0 &= \frac{N}{2} \\
x_{k+1} &= \frac{1}{2}\left(\frac{N}{x_k} + x_k\right) \\
&\vdots \\
x_{k+n} &\approx \sqrt{N}
\end{aligned}
$$

```c
#include <stdio.h>
#include <math.h> // for fabs funtion

// defining to what point of accuracy the Newton Raphson method will run
#define ACCURACY 0.001

int main(){
  setbuf(stdout, NULL);

  // float which stores the user's input
  float input;
  // asking the user for an input
  printf("What number would you like to compute the square root of\n");
  // accepting the user's input
  scanf("%f", &input);

  // Declare three floats which will store:
  // x0 - what the previous approximation was
```

```c
  // x1 - a more accurate approximation (current approximation)
  // error - a measure of how much the result has changed from the last iteration
  float x0, x1, error;

  // First Guess
  x1 = input/2;
  do{
    // what was the most recent approximation is set to the previous approximation
    x0 = x1;
    // Newton Raphson Algorithm
    x1 = (0.5)*(x0 + input/x0);
    // computing the magnitude of how much the approximation has changed from the previous iteration
    error = fabs(x1-x0);
    // do this until the error is less than the specified amount
  }while(error > ACCURACY);

  printf("The exact square root of %f is %f\n", input, sqrt(input));
  printf("The square root of %f is approximately %f (to the nearest %f)\n", input, x1, ACCURACY);

  return 0;
}
```

Table 5.1: Testing: Question 5

Input	**Exact Root**	**Approximated Root**
25	5	5.000000
65	8.062258	8.062258
99.552	9.977575	9.977575
1	1	1.000000
852639.32546	923.384704	923.384705
222111.4862	471.287051	471.287048

Question 6

This problem was completed successfully and works as intended.

```c
#include <stdio.h>
#include <stdlib.h> // for rand function
#include <time.h> // for time

// setting the number of elements in a row/column of the square matrices
#define SIZE 32

// setting the range of the random numbers [1 , range]
#define RANGE 999

int main() {
  setbuf(stdout, NULL);
  // setting the seed for random numbers to time
  srand(time(NULL));

  // declaring the 2D arrays of floats which will be used as matrices
  double matrixA[SIZE][SIZE];
  double matrixB[SIZE][SIZE];
  double matrixMULT[SIZE][SIZE];

  // counters for the for-loop to fill the matrices with random values, and to carry out the multiplication
  int i, j, k;

  // filling the matrices A&B with random Real Numbers
  for(i=0; i<SIZE; i++){
    for(j=0; j<SIZE; j++){
      matrixA[i][j] = (double) rand() / (double) (RAND_MAX/RANGE);
      matrixB[i][j] = (double) rand() / (double) (RAND_MAX/RANGE);
```

```
    }
  }

  // displaying the matrices
  printf("Matrix A:\n");
  for(i=0; i<SIZE; i++){
    for(j=0; j<SIZE; j++){
      printf("%f", matrixA[i][j]);
      // after every column insert a tab space for spacing
      printf("\t");
    }
    // after every row start a new line for spacing
    printf("\n");
  }
  printf("\nMatrix B:\n");
  for(i=0; i<SIZE; i++){
    for(j=0; j<SIZE; j++){
      printf("%f", matrixB[i][j]);
      printf("\t");
    }
    printf("\n");
  }

  // carrying out the multiplication
  double sum = 0;
  // first two nested 'for's which go through every element in the final array
  for(i=0; i<SIZE; i++){
    for(j=0; j<SIZE; j++){
      // setting the current element of the matrix holding the result of the multiplication to 0
      matrixMULT[i][j] = 0;

      // in this for loop, the program moves through the first array column by column,
      // and the second array row by row, each time working the product, and calculating
      // the value of the current element as a sum of each product
      for(k=0; k<SIZE; k++){
        sum += matrixA[i][k] * matrixB[k][j];
      }
      // setting the value of the current element in the matrix to the calculated sum
      matrixMULT[i][j] = sum;

      // resetting the sum
      sum = 0;
    }
```

```
  }

  // Displaying the result of the multiplication
  printf("\nResult of Multiplication:\n");
  for(i=0; i<SIZE; i++){
    for(j=0; j<SIZE; j++){
      printf("%f", matrixMULT[i][j]);
      printf("\t");
    }
    printf("\n");
  }

  return 0;
}
```

Testing: Question 6

The implementation of the matrix multiplication was modified, such that instead of being 32x32 matrices, they were made to be 3x3 matrices. This was done in order to make verification of the result easier. Such a method of testing was only functional since the code works for any two equally sized square matrices. The range of possible values was also reduced to 99 to avoid larger numbers.

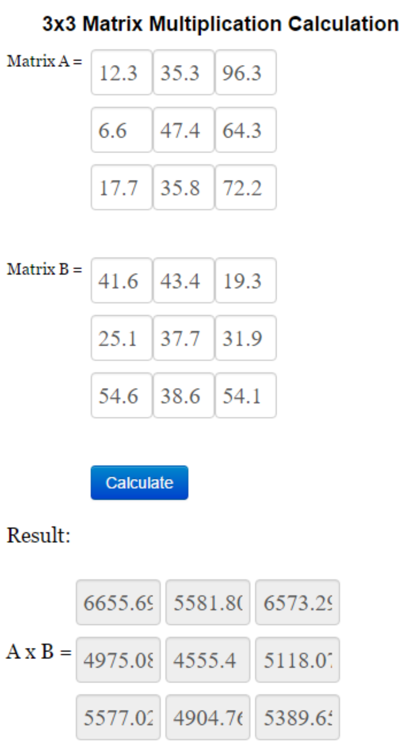
The code was run and the following was the output of the program:

```
Matrix A:
12.333333   35.254545   96.327273
6.560606    47.427273   64.257576
17.712121   35.751515   72.187879

Matrix B:
41.609091   43.351515   19.339394
25.142424   37.727273   31.890909
54.630303   38.581818   54.090909

Result of Multiplication:
6661.951625 5581.207860 6573.248448
4975.828301 4552.887971 5115.127677
5579.510716 4901.794068 5387.397998
```

The randomly generated input matrices were multiplied using an online tool to verify that the final result was in fact correct. For this http://ncalculators.com/matrix/3x3-matrix-multiplication-calculator.htm was used.



There were some discrepancies between the answer obtained by my implementation of the multiplication and that produced by the website since the values inputted to the website were only accurate up to 1 decimal place and there may have been some rounding errors during the calculation of the elements of the resulting matrix.

Question 7

This problem was completed successfully and works as intended.

```c
#include <stdio.h>

int getMax(int array[], int i, int max);

int main(){
  setbuf(stdout, NULL);

  int size;

  /*
    printf("Enter the size of the array: ");
    scanf("%d",&size);

  int array[size];

    printf("Enter %d elements of an array: ", size);
    for(i=0;i<size;i++)
      scanf("%d",&arr[i]);

    */

  int array[] = {-7, -68, -95, 0};
  size = sizeof(array)/sizeof(array[0]);

    int max = getMax(array, 0, size-1);

    printf("Largest element of the array is: %d", max);

    return 0;
}

// recursive function to find the largest element in a list
```

```
// parameters of this function are a pointer to the array,
  the start of the array, and the end of it
int getMax(int array[], int start, int end){
  // int which will store the largest element found
  int max;
  // if there is only one element being considered, then it
    is the largest one
  if(start == end){
    return array[end];
  }
  // calling the recursive function and starting from the
    next element
  max = getMax(array, start+1, end);
  // if the first element being considered is greater than
    the max, return it
    if(array[start] > max){
      return array[start];
    }
    // else return the current max
    else{
      return max;
    }
}
```

Table 7.1: Testing: Question 7

Input	**Expected Output**	**Actual Output**
25, 67, 35, 21, 12	67	67
25, -67, -35, 21, 12	25	25
-65, -32	-32	-32
-7, -68, -95, 0	0	0

Question 8

This problem was completed successfully and works as intended. The only problem that may arise is if the user enters a large number of terms to be calculated, as this sometimes cause overflows in the variables.
The following image was used as a guide to model the program to the Taylor Series of Trigonometric Functions.

$$\begin{aligned}\sin x &= x - \frac{x^3}{3!} + \frac{x^5}{5!} - \frac{x^7}{7!} + \cdots \\ &= \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n+1}}{(2n+1)!}, \\ \cos x &= 1 - \frac{x^2}{2!} + \frac{x^4}{4!} - \frac{x^6}{6!} + \cdots \\ &= \sum_{n=0}^{\infty} \frac{(-1)^n x^{2n}}{(2n)!}.\end{aligned}$$

```
#include <stdio.h>
#include <math.h> // for pow function
#include <stdlib.h>

// declaring the constant PI
#define PI 3.14159265359

float sin_cos(char fn, float input, int terms);
long factorial(int n);  // factorial function

int main(void) {
  setbuf(stdout, NULL);

```

```c
// char which holds whether the user wants to determine the sin or cos of a number
char trig_fn;
// float which holds the number on which trig function will be evaluated
float input;
// int which holds how many terms of the series expansion the user wants evaluated
int terms;
// float which holds the final answer of the calculation
float answer;

// determining whether the user wants to evaluate a sin or a cos
printf("Please enter whether you want to evaluate a sin or cos (s for sin or c for cos).\n");
scanf("%c", &trig_fn);

// determining on what value the user wants to compute the trig fn.
printf("Please enter the number you would like to compute the trig. fn. of. (in radians)\n");
scanf("%f", &input);
// calculating the modulus of the input to get it in the range of -2pi<input<2pi
input = fmod(input, 2*PI);
printf("%f\n", input);

// determining how many terms of the series expansion the user wants considered
printf("How many terms of the series expansion of the trigonometric function would you like to be evaluated?\n");
scanf("%d", &terms);

// calling the method sin_cos and passing whether the fn. is a sin or cos,
// the input, and the number of terms requested as arguments.
answer = sin_cos(trig_fn, input, terms);

if(trig_fn == 's'){
  printf("sin(%f) = %f (up to %d terms)", input, answer, terms);
}
else if(trig_fn == 'c'){
  printf("cos(%f) = %f (up to %d terms)", input, answer, terms);
}
```

```
  return 0;
}

float sin_cos(char fn, float input, int terms) {

  // float which will store the answer of the input
  float ans = 0.0;

  // counter for loop for series
  int i;

  // if the user wants a sin function evaluated
  if (fn == 's') {
    for (i = 0; i < terms; i++) {
      ans += pow(-1, i) * pow(input, (2*i + 1)) / factorial(2*i + 1);
    }
  } else if (fn == 'c') {
    for (i = 0; i < terms; i++) {
      ans += pow(-1, i) * pow(input, (2*i)) / factorial(2*i);
    }
  }

  return ans;
}

// function to evaluate factorials
long factorial(int n) {
  if (n == 0)
    return 1;
  else
    return (n * factorial(n - 1));
}
```

Table 8.1: Testing: Question 8

Input (sin/cos, radians, terms)	**Exact Value**	**Calculated Value**
c, 0, 3	1	1
s, 0, 5	0	0
c, 3.14, 5	-0.9999	-0.9761
s, 6.59, 7	0.3020	0.3020
c, -13.24, 9	0.7816	0.7816
s, -13.24, 11	-0.6238	-0.6238
c, 25.19, 15	0.9984	0.9984

Question 9

This problem was completed successfully and works as intended.

```c
#include <stdio.h>
#include <stdlib.h> // for exit function

long fibonacci(int term);

int main() {
  setbuf(stdout, NULL);

  // int which will hold how many terms of the Fibonacci Series the user wants to calculate
  int term;
  // ask the user how many terms he would like calculated
  printf("Till which term of the fibonacci sequence do you wish to calculate\n");
  // accept the number of terms as an input
  scanf("%d", &term);

  // output the sum of the terms up until the term requested by the user
  printf("The sum of the fibonacci series up to the %dth term is %ld\n", term, fibonacci(term));

  return 0;
}

long fibonacci(int term){

  // Special cases
  if (term <= 0){
    printf("The requested term is invalid (must be greater than or equal to 1)\n");
    exit(0);
  }
```

```
  else if(term > 45){
    printf("The following answers are invalid due to the capacity of the long data type\n");
    printf("Try calculating up to a term less than 46\n");
  }
  if(term == 1){
    return 0;
  }

  // General Method

  // create an array of ints with as many elements as terms were requested by the user
  // to hold each term of the fibonacci sequence
  long fibonacci[term];
  // set the first term of the fibonacci sequence to 0
  fibonacci[0] = 0;
  // set the second therm of the fibonacci sequence to 1
  fibonacci[1] = 1;
  // declare the int which will hold the sum of the terms entered so far
  // initialise it to the sum of the first and second term
  long sum = fibonacci[0] + fibonacci[1];

  // declare a counter for loops
  int i;
  // for loop which starts from the third term of the fibonacci sequence up until the term entered by the user
  for (i = 2; i < term; i++) {
    // set the value of the current term to the sum of the previous 2 terms
    fibonacci[i] = fibonacci[i - 1] + fibonacci[i - 2];
    // add the current term to the sum of all the terms of the series
    sum += fibonacci[i];
  }

  // for loop which displays all the terms of the Fibonacci Sequence
  for (i = 0; i < term; i++) {
    printf("%ld\t", fibonacci[i]);
  }
  printf("\n");

  return sum;
}
```

Table 9.1: Testing: Question 9

Input	**Expected Output**	**Actual Output**
0	Invalid Input	Invalid Input
1	0	0
2	1	1
3	2	2
5	7	7
10	88	88
25	121392	121392
45	1836311902	1836311902
46	The following answers are invalid due to the capacity of the long data type Try calculating up to a term less than 46	The following answers are invalid due to the capacity of the long data type Try calculating up to a term less than 46