Data Structures and Algorithms

2017/2018

Name: Valerija Holomjova
Course Code: ICS1018-SEM2-A-1718

Table of Contents

Question 1

This code is fully functional. At the start of the code, a macro 'N' was defined to represent the number of elements in the array. Similarly, an array 'ar' was defined to hold 'N' number of integers. For this question, it has been assumed that these values are to be predefined according to the user's preference. In this example, the value of 'N' is 7 whilst the integers in the array were chosen at random.
A 'for' loop is used to go through each element in the array and find its product pair using the 'pairs()' function. This function accepts one argument which will represent the value to which the pairs will be found. In other words, it is the product of the final pairs and is represented as the integer 'num' in the function. In the 'pairs()' function, two 'for' loops are used to find possible pair factors for the product 'nu' up to a maximum of 1024 integers. The letters 'a', 'b', 'c', 'd' represent the possible factors with 'a' and 'b' being one pairs and 'c' and 'd' being another. In addition, an integer 'found' was defined to keep tracks of the number of pairs found at the end of the 'for' loops. For instance, the value '1' would have 0 pairs, therefore, 'found' would be equivalent to 0 at the end of the function.

In the 'for' loops, a number of 'if' statements are used to ensure that the values of 'a', 'b', 'c' and 'd' are not equal if they are found. The first 'if' statement, checks whether two numbers in the 1024x1024 search space that are factors of the product 'num' were found and are not equal.If this condition is met, another 'if' statement is used to check whether 'a' and 'b' or 'c' and 'd' have been found yet to avoid overlapping the previous factors found. If the first pair ('a' and 'b') has not been found yet, the newly found factors will be set to 'a' and 'b'. Elsewise, if 'a' and 'b' were already found, the newly found factors will be checked to see whether their values are equivalent to the previous found pair, if not, they will represent the second found pair, 'c' and 'd'.

At the end of the function, 'if' and 'else if' statements are used to output the correct message depending on how many product pairs were found for the number 'num' and which ones they were. It is important to note that this function will only find up to maximum of two product pairs for each number. Please refer to the diagram below which illustrates the expected outcome and actual outcome of the program when executed.

Predefined Input	Expected Output/Outcome	Actual Output/Outcome
#define N 7 int ar[N] = {1,2,24,120,450,876,1016};	"No pairs were found for the product 1. Only one pair of integers have the product 2: 1 * 2 = 2 The two pairs of integers that have the same product 24 are: 1 * 24 = 24 2 * 12 = 24 The two pairs of integers that have the same product 120 are: 1 * 120 = 120 2 * 60 = 120 The two pairs of integers that have the same product 450 are: 1 * 450 = 450 2 * 225 = 450 The two pairs of integers that have the same product 876 are: 1 * 876 = 876 2 * 438 = 876 The two pairs of integers that have the same product 1016 are: 1 * 1016 = 1016 2 * 508 = 1016"	"No pairs were found for the product 1. Only one pair of integers have the product 2: 1 * 2 = 2 The two pairs of integers that have the same product 24 are: 1 * 24 = 24 2 * 12 = 24 The two pairs of integers that have the same product 120 are: 1 * 120 = 120 2 * 60 = 120 The two pairs of integers that have the same product 450 are: 1 * 450 = 450 2 * 225 = 450 The two pairs of integers that have the same product 876 are: 1 * 876 = 876 2 * 438 = 876 The two pairs of integers that have the same product 1016 are: 1 * 1016 = 1016 2 * 508 = 1016"

Please refer below for the code of Question 1:

```c
#include <stdio.h>
//N is the number of elements in the array.
#define N 7
void pairs(int num);

int main() {
    //Initializing an array with N elements.
    int ar[N] = {1,2,24,120,450,876,1016};

    //Finds the pairs of each number in the array using the pairs() function.
    for(int i = 0; i < N; i++) {
        pairs(ar[i]);
    }
}

//Function that finds the pair of any given number passed as 'num'.
void pairs (int num) {
    int found = 0; //The number indicates how many pairs were found eg. 0=0 pairs
found.
    int a = 0, b = 0, c = 0, d = 0;
    int i,j;

    //The For loops scan for factors of the product 'num' up to a value of 1024.
    for(i=0; i<1024; i++) {
        for(j=0; j<1024; j++) {
            //If statement checks whether two numbers in the search space are
factors of 'num' and are not equal.
            if(((i*j) == num) && (i!=j)) {
                    //Executes if a and b haven't been found yet.
                    if(a==0 && b==0) {
                        a = i;
                        b = j;
                        found = 1;
                        //Executes if c and d haven't been found yet.
                    } else if (c==0 && d==0) {
                        //If statement checks that i and j are not the same as a
and b.
                        if((a!=i) && (a!=j) && (b!=i) && (b!=j)) {
                            c = i;
                            d = j;
                            found = 2;
                        }
                    }
                }
            }
        }

    //Illustrates which pairs that were found for 'num'.
    if(found == 0) {
        printf("No pairs were found for the product %d.\n", num);
    } else if (found == 1) {
        printf("Only one pair of integers have the product %d: \n", num);
        printf("%d * %d = %d \n", a, b, num);
    } else if (found == 2) {
        printf("The two pairs of integers that have the same product %d are: \n",
num);
        printf("%d * %d = %d \n", a, b, num);
        printf("%d * %d = %d \n", c, d, num);
    }
}
```

Question 2

This code is fully functional. At the start of the code I defined a macro called ‘MAX’ which will represent the maximum amount of data elements in the stack, as well as the maximum amount of characters that will be read from the RPN expression that the user inputs. For this question the value of ‘MAX’ has been predefined to a value of 100, however, this can be changed according to the user’s preference.

Next, an array of double variables is created which represents the ADT stack and has the name ‘stack’. The integer ‘index’ is initialized to a value of 0 and keeps track of the top of the stack, hence, every time an element is added to the stack, the value of ‘index’ will increment by 1. Both variables have been initialized outside the functions to be used as global variables.

In the main function, the user is prompted to input an RPN expression to evaluate. The input is read from the user through standard input and stored in the string ‘s’ using the library function ‘fgets()’. Next, the library function ‘strtok()’ is used to split the string into tokens which are sequences of characters in between a specified delimiter. In this case, the delimiter was set to the space character.

The character pointer ‘tok’ will point to one of the tokens at the time. Before the ‘while’ loop, the first token is stored in the pointer ‘tok’. It is then passed through a series of ‘if’ statements to determine whether it is a number or an operator. First, an ‘if’ statement checks whether the token is a number using the library ‘isdigit()’ function. If it is a number, it is first converted into a double variable using the library function ‘atof()’ and then pushed onto the top of the stack using the ‘push()’ function. If it is an operator, two elements are popped from the stack using the ‘pop()’ function and evaluated based on the type of operator. It is important to note that for the subtraction and division element, a variable ‘x’ is assigned the first popped element as these two operations are not commutative.

The first character of the token is only considered in the ‘if’ statement, thus, it is important that the user inputs valid RPN expressions. If the user inputs characters that are not operators or numbers, then the integer ‘flag’ will be set to 1, indicating that the user input an invalid expression. The next token is gathered at the end of the ‘while’ loop using the ‘strtok()’ function which will terminate once the value of ‘tok’ is ‘NULL’. This will occur when all the tokens from the input have been passed through the ‘while’ loop. After the termination of the ‘while’ loop, the final value (by 2 decimal points) is displayed to the user by popping the current top element in the stack. The final value is only displayed if the integer ‘flag’ is equivalent 0, indicating that the expression consisted of valid characters.

The ‘push’ function works by passing a double variable through as an argument, which is then represented as ‘n’. This value is then placed at the top of the stack which is kept track of using the variable ‘index’. The value of ‘index’ is incremented by 1 as another element has been added to the stack. The ‘pop()’ function works by returning the value that is at the top of the stack. The value of ‘index’ is then decreased by 1 as the element is removed from the stack.

It is essential to note that if the user does not input spaces between each operand and operator, the code will not function correctly as the space character is used as a delimiter. Moreover, the code will not pick up traces of invalid characters if they are placed alongside a valid character (eg. 2abc) as the ‘if’ statements only consider the first character of each token. Please refer to the diagram on the

next page which illustrates the expected outcome and actual outcome of the program when executed. Please refer to the next page for the code of Question 2.

Input	Expected Output/Outcome	Actual Output/Outcome
5 5 + 6 * 10 / 200 +	"Final value is 206.00.	"Final value is 206.00."
5 2 / 7 * 550 + 246 -	"Final value is 321.50."	"Final value is 321.50."
55 5 +6 * (no space between the + and 6)	Expression not properly evaluated.	"Final value is 0.00."
5 5 + abc * (invalid expression)	"Error. Wrong Input."	"Error. Wrong Input."

```c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
#define MAX 100

void push(double n);
double pop();

double stack[MAX]; //An array representing the stack.
int index = 0; //Keeps track of the top of the stack.

int main() {
    char s[MAX],*tok;
    const char space[2] = " ";
    double x;
    int flag = 0;
    printf("Please input the RPN expression.\n");
    fgets(s,MAX,stdin); //Scans the user's input and stores it in the string 's'.
    tok = strtok(s,space); //Stores the first token of the string 's' in 'tok'.
    while( tok != NULL) {
        if(isdigit(tok[0])) {
            push(atof(tok)); //Pushes the number on top of the stack.
            //The next statements evaluates an expression depending on the operator
and pushes it on the stack.
        } else if(tok[0] == '+'){
            push(pop() + pop());
        } else if(tok[0] == '-'){
            x = pop();
            push(pop() - x);
        } else if(tok[0] == '/'){
            x = pop();
            push(pop() / x);
        } else if(tok[0] == '*'){
            push(pop() * pop());
        } else  //Executes if the input isn't a number or operator.
            flag = 1;
        //Stores the remainder tokens of the string 's' in 'tok' until NULL is
reached.
        tok = strtok(NULL, space);
    }
    if(flag == 0)
        printf("Final value is %.2lf.", pop());
    else
        printf("Error. Wrong Input.\n");
}

//Function that inserts an element at the top of the stack.
void push(double n){
    stack[index++] = n;
}

//Function that removes an element at the top of the stack.
double pop(){
    return stack[--index];
}
```

Question 3 Part 1

This code is fully functional. At the start of the code, the user is asked to input a number to check whether it is prime. The input is scanned and stored in the integer 'num'. An 'if' statement is used to display a message as to whether the number is prime or not depending on the value 'num' returns after passing through the 'prime()' function.

The 'prime()' function is of type Boolean so it returns a value or true or false. The integer 'num' is the only argument of the function and represents the value which will be checked for whether it is prime or not. In the function, the value 'num' is first passed through an 'if' statement which checks whether the number is 0 as it is not considered a prime number and the upcoming 'for' loop will only determine non-prime numbers from the value of 2 and above. If the variable 'num' holds the value of 0, the function will return a value of false indicating that the number is not prime. Otherwise, it will fall into the 'for' loop which determines whether 'num' is non-prime by checking whether it is divisible by any number between 2 and the square root of 'num'. The 'for' loop starts checking for divisible numbers from the starting value of 2 as numbers that are divisible by 1 can be considered prime. An optimization has been made so that the 'for' loop only checks for numbers till the value of it's square root as any factors beyond the square root would be paired up with numbers below it's value, hence it's lesser pair would have already been found by the 'for' loop. This allows the algorithm to be faster. If a value between 2 and the square root of 'num' is divisible by 'num', which is checked using the modulus operator, the function will return a value of false, indicating that 'num' is not a prime number as it has a factor other than the value of 1 and itself. Otherwise, the 'for' loop will terminate and the function returns a value of true, indicating that no factors were found, hence the number is prime. Please refer to the diagram below which illustrates the expected outcome and actual outcome of the program when executed. Please refer below for the code of Question 3 Part 2.

Input	Expected Output/Outcome	Actual Output/Outcome
0	"0 is NOT a prime number."	"0 is a NOT prime number."
6	"6 is NOT a prime number."	"6 is NOT a prime number."
1453	"1453 is a prime number."	"1453 is a prime number."

```
#include <stdio.h>
#include <stdbool.h>
#include <math.h>
bool prime(int num);

int main() {
    int num, i;
    printf("Please input a number to check whether it is prime.\n");
    scanf("%d",&num);
    if(prime(num) == true)
        printf("%d is a prime number.\n",num);
    else
        printf("%d is NOT a prime number.\n",num);
}

//Function that returns true or false based on whether 'num' is prime or not.
bool prime(int num) {
    if(num == 0){
        return false; //if the number is 0, it is not prime.
    } else {
        for (int i = 2; i <= sqrt(num); i++) {
            if (num % i == 0)
                return false; //if the number is divisible by any number between 2 and it's
square root, it is not prime.
        }
    }
    return true;
}
```

Question 3 Part 2

This code is fully functional. At the start of the code, the user is asked to input a number to check whether it is prime. The input is scanned and stored in the integer 'num'. An 'if' statement is used to display a message as to whether the number is prime or not depending on the value 'num' returns after passing through the 'prime()' function.

The 'prime()' function is of type Boolean so it returns a value or true or false. The integer 'num' is the only argument of the function and represents the value which will be checked for whether it is prime or not. In the function, the value 'num' is passed through a series of 'if' statements to determine whether it is prime. The first one checks whether the number is 1 as it is not considered a prime number. Next the 'if' statement checks whether 'num' is the value 2,3,5, or 7 as these are all basic prime numbers. The following 'if' statement is an implementation of the Sieve of Eratosthenes Algorithm. In his algorithm, multiples of 2,3,5,7 are cancelled out from a grid of numbers and the remaining numbers are considered prime numbers. Thus, through the 'if' statements, numbers that are not 1 and are not divisible by 2,3,5,7 (except 2,3,5,7 themselves) can be considered prime numbers. This method was implemented as I felt that it was much more efficient than looping a number 'n' for 'n-1' times for larger numbers. Please refer to the diagram below which illustrates the expected outcome and actual outcome of the program when executed. Please refer below for the code of Question 3 Part 2.

Input	Expected Output/Outcome	Actual Output/Outcome
2	"2 is a prime number."	"2 is a prime number."
6	"6 is NOT a prime number."	"6 is NOT a prime number."
1453	"1453 is a prime number."	"1453 is a prime number."

```c
#include <stdio.h>
#include <stdbool.h>
bool prime(int num);

int main() {
    int num, i;
    printf("Please input a number to check whether it is prime.\n");
    scanf("%d",&num);

    if(prime(num) == true) {
        printf("%d is a prime number.\n",num);
    } else {
        printf("%d is NOT a prime number.\n",num);
    }
}

//Function that returns true or false based on whether 'num' is prime or not.
bool prime(int num) {
    //If 'num' is 1 it is not considered a prime number.
    if(num == 1) {
     return false;
        //If 'num' is 2,3,5,7 it is a prime number.
    } else if(num == 2 || num == 3 || num == 5 || num == 7) {
        return true;
        //Checks if 'num' is divisible by 2,3,5,7 - implementation of Sieve of
Eratosthenes Algorithm.
    } else if(num % 2 == 0 || num % 3 == 0 || num % 5 == 0 || num % 7 == 0) {
        return false;
    } else {
        return true;
    }
}
```

Question 4

This code is fully functional. At the start of the code, a 'struct' statement is created called 'node' which will represent each node in the BST. Its members hold the information of each node, specifically, the value of the node as well as a pointer to its left and right subtrees.

In the main function, the user is asked to enter values repeatedly. The values are scanned using a while function which will terminate when a non-integer is inputted. Each value is stored in the integer 'num' and is passed through the function 'insert()' along with the address of the root node denoted as 'start'. The 'insert()' function is used to insert each value into the BST. It is important to note that 'start' was initialized to a value of NULL before the 'while' loop to indicate that no values have been inputted yet and avoid bugs.

In the 'insert()' function two pointers 'cn' and 'pn' of type struct node are defined and represent the current node and previous node respectively. These will need to be kept of track of while looking for an available position for the value 'num' in the BST. An 'if' statement is used to check whether the root node has a value in its place. If it is empty (or NULL), the 'new()' function will be used to input the value of 'num' into the root node. The 'new()' function will be described following the description of this function. If the root node already has a value, a 'while' loop is used to locate an empty node as it terminates when the current node 'cn' is equivalent to NULL. Until this empty node is found, it checks whether the user's inputted value 'num' is greater or smaller than the number stored in the current node. If 'num' is greater, the value of 'cn' is set to the value of its right subtree. If 'num' is smaller, the value of 'cn' is set to the value of the left subtree. This process repeats until the current node is empty. A Boolean variable called 'left' is used to determine whether current node was accessed through the left or right subtree of the previous node. Thus, once the 'while' loop terminates, a new node with the 'num' value is created through the 'new()' function and is stored in the left or right subtree of the previous node depending on whether 'left' is true or false. The root is always returned so that the program could start searching for available spaces from the start of the BST.

The 'new()' function is used to create nodes. This is done by creating a pointer to a node called 'new' which is allocated memory using the library function 'malloc()'. Next, the 'num' value is stored in the node. Next, the left and right subtrees are initialized to NULL as the node is new and has no children yet. The address of the newly created node is then returned. Please refer to the table below for the expected outcome and actual outcome of the program when executed.

Input	Expected Output/Outcome	Actual Output/Outcome
1	1 is stored in the 'num' value of 'start'.	1 is stored in the 'num' value of start.
5,2,7	5 is stored in the 'num' value of 'start'. The 'left' of 'start' points to the node containing the value of 2. The 'right' of 'start' points to the node containing the value of 7.	5 is stored in the 'num' value of 'start'. The 'left' of 'start' points to the node containing the value of 2. The 'right' of 'start' points to the node containing the value of 7.
5,2,3	5 is stored in the 'num' value of 'start'. The 'left' of 'start' points to the node containing the value of 2. The 'left' of this node (containing the value of 2) points to a node containing the value of 3.	5 is stored in the 'num' value of 'start'. The 'left' of 'start' points to the node containing the value of 2. The 'left' of this node (containing the value of 2) points to a node containing the value of 3.

Please refer to the next page for the code of Question 4.

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

//Struct that stores information of each node in the BST.
struct node{
    int num; //Value of the node.
    struct node *left; //Points to the left subtree.
    struct node *right; //Points to the right subtree.
};

struct node *insert(struct node *start, int num);
struct node *new (int num);

int main() {
    int num;
    printf("Please enter a sequence of integers or press 'q' to quit.\n");
    //The starting node is defined and set to NULL to avoid bugs.
    struct node *start = NULL;
    //While loop used to scan user input one by one and store it in the BST.
    while(scanf("%d",&num) == 1){
        start = insert(start,num);
    }
}

//Function that inserts the user's inputted value into the BST.
struct node *insert(struct node *start, int num) {
    //cn points to the current node, pn points to the previous node.
    struct node *cn = start, *pn = NULL;
    bool left = false;
    //If the node is empty - creates a node with the user's inputted value and
returns it.
    if(start == NULL) {
        return new(num);
    }
    /*If the user's inputted value is larger than the current's node value, insert
it into the right subtree.
     * If the user's inputted value is smaller than the current node's value,
insert it into the left subtree.*/
    while(cn != NULL) {
        pn = cn;
        if (num >= cn->num) {
            cn = cn->right;
            left = false;
        } else {
            cn = cn->left;
            left = true;
        }
    }
    //To check check if the cn should be pointed to by the pn's left or right
subtree.
    if(left == false){
        pn->right = new(num);
    } else {
        pn->left = new(num);
    }
    return start;
}

//Function that creates the actual nodes.
struct node *new (int num) {
    //Allocates memory for a new node and defines it's values.
    struct node *new = (struct node *) malloc(sizeof(struct node));
    new->num = num;
    new->left = NULL;
    new->right = NULL;
    return new;
}
```

Question 5

This code is fully functional. At the start of the code, the user is asked to input a value 'num' for which the square root will be approximated. The float variable 'square' will represent the square root of the value 'num' and is obtained through the 'squareroot()' function which accepts a guess and 'num' as its arguments. The square root of the user's inputted number 'num' is then printed.

The first three functions are all basic math functions.

- The first function accepts a number and returns its value squared.
- The second function returns the average of two numbers using basic mathematic principles.
- The third function returns the absolute of a number. In other words, the function returns a positive value of the number inputted.

To approximate the square root, I have implemented the Babylonian Square Root method into the code. It works by taking a guess, dividing the original number by the guess and finding the average of these numbers. The value of the average is then used as the next guess. In the function, an if statement is used to check the accuracy of the current guess. The error percentage is determined by squaring the guess and subtracting it by the original number 'num'. If the error percentage is below 0.0001, the guess is returned as it is very close to the actual square root. Otherwise, the guess is divided by the original number and the average of this value and the guess itself is obtained to be used as the next guess by passing it through the 'squareroot()' function again. Note that the 'num' value passed through must always remain the same as it denotes the original number the user inputted.

Please refer to the table below for the expected outcome and actual outcome of the program when executed.

Input	Expected Output/Outcome	Actual Output/Outcome
1	The square root of 1.000 is 1.000.	The square root of 1.000 is 1.000.
4	The square root of 4.000 is 2.000.	The square root of 4.000 is 2.000.
36	The square root of 36.000 is 6.000.	The square root of 36.000 is 6.000.
120	The square root of 120.000 is 10.954.	The square root of 120.000 is 10.954.

Please refer to the next page for the code of Question 5.

```c
#include <stdio.h>
float average(float num1, float num2);
float absolute(float num);
float squareroot(float guess, float num);

int main(){
    float num;
    printf("Please enter a number to approximate it's square root.\n");
    scanf("%f",&num);
    float square = squareroot(1,num);
    printf("The square root of %.3f is %.3f.\n",num,square);
}

//Function that returns a number squared.
float squared(float num) {
    return num * num;
}

//Function that returns the absolute of two numbers.
float average(float num1, float num2) {
    return (num1+num2)/2;
}

//Function that returns the absolute of a number.
float absolute(float num) {
    if(num<0){
        return -num;
    } else {
        return num;
    }
}

//Function that returns the square root of a number.
float squareroot(float guess, float num) {
    /*Checks the accuracy of the current guess.
    If the error percentage of the guess is less than 0.0001, the guess is
returned*/
    if(absolute(squared(guess)-num)<0.0001){
        return guess;
    } else {
        //Recalls the function after applying the formula.
        guess = squareroot(average(guess,(num/guess)),num);
    }
    return guess;
}
```

Question 6

This code is fully functional. At the start of the code, a macro 'N' was defined to represent the number of elements in the array. Similarly, an array denoted as 'array' was defined to hold 'N' number of integers. For this question, it has been assumed that these variables are to be predefined according to the user's preference. In this example, the value of 'N' is 15 whilst the integers in the array were chosen at random.

In the code, two 'for' loops are used to go through each element in the array, and compare it to the succeeding elements. The second loop checks only succeeding elements to increase the efficiency of the program instead of checking for repeated values from the start of the array as those should have already been inspceted. An 'if' statement is used to check whether the value of the element has been repeated at any stage in the array by using equivalence. The element must also not have the value of 'NULL'. This is due to the reason that once an element is found to be repeated, its value is displayed to the user and all elements in the array with the same value including itself are set to 'NULL'. This is done to avoid double counting values. The limitation of this code is that it does not display how many times the integer is repeated. However, this can be achieved by modifying the code using flags. Please refer to the table below for the expected outcome and actual outcome of the program when executed.

Predefined Input	Expected Output/Outcome	Actual Output/Outcome
#define N 15 int array[N] = {1,1,5,4,3,9,9,100,45,45,3,4,1,2,5};	"The following numbers are repeated more than once: 1 5 4 3 9 45"	"The following numbers are repeated more than once: 1 5 4 3 9 45"
#define N 15 int array[N] = {1,100,58,4,3,9,9,100,45,45,3,4,1,20,58};	"The following numbers are repeated more than once: 1 100 58 4 3 9 45"	"The following numbers are repeated more than once: 1 100 58 4 3 9 45"
#define N 20 int array[N] = {1,100,58,48,6,9,9,10,45,45,3,4,1,20,58,58,6,1,46,101}	"The following numbers are repeated more than once: 1 58 6 9 45"	"The following numbers are repeated more than once: 1 58 6 9 45"

Please refer to the section below for the code of Question 6.

```
#include <stdio.h>
//N is the number of elements in the array.
#define N 15
int main(){
    int array[N] = {1,1,5,4,3,9,9,100,45,45,3,4,1,2,5};
    printf("The following numbers are repeated more than once:\n");

    for (int i = 0; i < N; i++) {
        for (int j = i+1; j < N; j++) {
            //If the element is repeated in the array, and not null (already been
found).
            if((array[i] == array[j]) && (array[i] != '\0')) {
                //Prints the element that has been repeated.
                printf("%d ",array[i]);
                for (int k = j; k < N; k++) {
                    //Sets all the elements of same value in array to null
including itself.
                    if(array[k] == array[i]) {
                        array[k] = '\0';
                    }
                }
            }
        }
    }
}
```

Question 7

This code is fully functional. At the start of the code, a macro 'N' was defined to represent the number of elements in the array. Similarly, an array denoted as 'array' was defined to hold 'N' number of integers. For this question, it has been assumed that these variables are to be predefined according to the user's preference. In this example, the value of 'N' is 7 whilst the integers in the array were chosen at random. Next, the user is notified of the maximum number in the array which is found using the 'max()' function.

The 'max' function finds the maximum integer in the array by working backwards from the last element of the array and keeping track of the maximum so far until the first element is reached and the maximum is then returned. Thus, the argument 'array[]' represents the address of the array, the integer 'n' represents the current element being checked for whether it is the maximum or not, and the integer 'x' represents the current maximum at each stage.

At the start of the 'max()' function is the stopping condition which is reached when the first element of the array 'array[0]' is reached which indicates the end of the search for the maximum as the array is checked backwards. If the element is larger than the current maximum 'x', then the element is returned. If the element is smaller, then 'x' is returned.

Until the stopping condition is reached, an 'if' statement checks whether the current element being checked 'array[n]' is greater or smaller than the current maximum 'x'. The function will be called, and the greater value will be passed as the maximum in the next recursive step. The search space is also decreased as 'n-1' is passed through instead of n, signifying that the next element is being checked. Please refer to the table below for the expected outcome and actual outcome of the program when executed.

Predefined Input	Expected Output/Outcome	Actual Output/Outcome
#define N 7 int array[N] = {2,6,10,3,5,8,1};	"The maximum number is 10."	"The maximum number is 10."
#define N 7 int array[N] = {23,6,10,3,50,80,1};	"The maximum number is 80."	"The maximum number is 80."
#define N 15 int array[N] = {23,6,10,3,50,80,1,76,20,100,120,3,1,45,99};	"The maximum number is 120."	"The maximum number is 120."

Please refer to the next page for the code for Question 7.

```c
#include <stdio.h>
//N is the number of elements in the array.
#define N 15
int max(int array[], int n, int x);

int main() {
    int array[N] = {23,6,10,3,50,80,1,76,20,100,120,3,1,45,99};
    printf("The maximum number is %d.\n", max(array, N-1, 0));
}

//Function that finds the maximum number in the array by checking it backwards.
//'n' is the size of the current search space (and which element),'x' is the
current maximum.
int max(int array[], int n, int x){
//when 'n' is 0, the last element in the search space (first element of array) is
being checked - stopping condition.
    if (n == 0) {
        if(array[n] > x){
            return array[n];
        } else {
            return x;
        }
    }
    //Updates the current maximum and checks the next element.
    if(array[n] > x){
        return max(array, n-1, array[n]);
    } else {
        return max(array, n-1, x);
    }
}
```

Question 8

This code is fully functional. At the start of the code, the user is asked to input a number of their choice for which the cosine value will be computed. Then the program proceeds to ask the user the number of terms the user you would like to compute in the expansion. Next, the user is displayed the cosine value for the number which is computed using the 'cosine()' function for an 'n' amount of terms. It is important to note that for this question, I have chosen to compute the cosine of a number. The code can easily be adapted to calculate the sine instead if preferred.

In the 'cosine' function, the value 1 is passed through as 'cosn' which represents the current value of the cosine and will be added onto in the upcoming 'for' loop. The value 'x' represents the number for which the cosine will be computed for. The value 'n' represents the number of terms for which the cosine will be computed for. The value 1 was used at it is the first term in the McLauren's Series Expansion of Cosine. A float variable 'numfact' is initialized to 2 and represents the value of the number in the denominator of each term without the factorial as well as the power on the 'x' value. It is set to 2 as the second term of the McLauren's Series Expansion of Cosine has a denominator of 2! And a numerator of x^2. Every time a term is added or subtracted in the 'for' loop, this value will be incremented by 2 as the difference between the denominator in each term in the series is (n+2)! Whilst the difference in powers is (n+2).

The 'for' loop is used to add the value of the next term in the cosine expansion up to 'n' times. The 'i' index represents the i^{th} term being computed and starts from the second term as the first term is already stored in 'cosn' after being passed through as an argument.

An 'if' statement is used within the loop to determine whether the term is to be added or subtracted to the current value of cosine. By observation, it has been noted that the even number of terms (i.e. the second or fourth term and so on) are negative, whilst the odd number of terms (i.e. the third or fifth term and so on) are positive. Hence, when the i^{th} term is not divisible by 2 (i % 2 != 0), the number of terms are odd and hence the term must be added onto the current cosine value. On the other hand, if 'i' is divisible by 2, the number of terms are even and hence the term must be added onto the current cosine value.

The value of the term is calculated using the formula "(pow(x, numfact) / factorial(numfact)" which was constructed based on the general form of each term in the McLauren's Series Expansion of Cosine. The library function 'pow()' will calculate the value of x to the power of 'numfact'. It is then divided by the factorial of 'numfact' which is calculated using the 'factorial()' function. After 20 terms have been computed, the value of 'cosn' is returned.

The 'factorial()' function is recursive and calculates the factorial of a number by multiplying the number 'num' by the factorial of 'num-1' until the stopping condition is met. The stopping condition returns the value of 1 as it is the last term in the expansion of a factorial. Please refer to the table below for the expected outcome and actual outcome of the program when executed.

Input	Expected Output/Outcome	Actual Output/Outcome
Number:5 Terms:20	"The cosine expansion of cos5 for 20 terms is 0.2837."	"The cosine expansion of cos5 for 20 terms is 0.2837."
Number: 10 Terms: 100	"The cosine expansion of cos10 for 100 terms is -0.8390."	"The cosine expansion of cos10 for 100 terms is -0.8390."
Number:20 Terms:50	"The cosine expansion of cos20 for 50 terms is 0.4081."	"The cosine expansion of cos20 for 50 terms is 0.4081."
Number: 50 Terms:100	"The cosine expansion of cos50 for 100 terms is -1.#IND."	"The cosine expansion of cos50 for 100 terms is 0.964".

As shown from the diagram, the issue with the program is that it is not capable of computing the expansion of cosine for larger values such as 50. Thus, this problem is limited to computing only smaller values of cosine. Please refer below for the code for Question 8.

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

float factorial(float num);
double cosine(double cosn, float x, int n);

int main(){
    float num;
    int n;
    printf("Please enter a number to compute cosine.\n");
    scanf("%f",&num);
    printf("Please enter a number of terms.\n");
    scanf("%d",&n);
    printf("The cosine expansion of cos%.0f for %d terms is
%.4lf.\n",num,n,cosine(1,num,n));
}

//Function that finds the cosine of a function.
double cosine(double cosn, float x, int n) {
    float numfact = 2; //'numfact' is the denominator of term (without factorial)
in series expansion.
    //Calculates the value of cosine by adding 20 terms of McLauren's series
expansion.
    for (int i=2; i<n;i++){
        //'i' is the ith term in series expansion. The if statement determines if
it is positive or negative.
        if(i % 2 != 0){
            cosn += (pow(x, numfact) / factorial(numfact));
        } else {
            cosn -= (pow(x, numfact) / factorial(numfact));
        }
        numfact += 2;
    }
    return cosn;
}

//Function that finds the factorial of a number.
float factorial(float num) {
    //Stopping condition - the last term in the factorial is 1.
    if(num == 1) {
        return 1;
    } else {
            return num * factorial(num-1);
        }
    }
```

Question 9

This code is fully functional. At the start of the code, the user is asked to input a number of the term to which the sum of the first 'n' numbers of the Fibonacci Sequence will be calculated. This value will be represented by the integer 'n'. The sum of the terms till the n^{th} term is then calculated using the 'sum()' function and the value is displayed to the user.

The 'sum()' function operates by adding each term of the Fibonacci sequence from the first term to the n^{th} term by calculating the value each term individually and accumulating it onto the value of the integer 'sum'. The value of each term will be calculated using the 'fib()' function. The integer 'sum' is then returned after the termination of the 'for' loop.

The 'fib' function calculates the value of the n^{th} Fibonacci term through recursion. It is adapted from the general Fibonacci formula for the nth number which is $F_n = F_{n-1}+F_{n-2}$. This denotes that the value of the nth term is the sum of the two previous terms. Hence, the function finds the value by trying to find the Fibonacci value of the previous two terms which leads to a recursive loop until the stopping condition is reached when the value of the first two Fibonacci terms are reached which are both 1. After backtracking, the value of the two previous terms are found and the value of the n^{th} Fibonacci term is found. Please refer to the table below for the expected outcome and actual outcome of the program when executed. Please refer below for the code of Question 9.

Input	Expected Output/Outcome	Actual Output/Outcome
5	"The sum of the 5 term of the Fibonacci is 12."	"The sum of the 5 term of the Fibonacci is 12."
10	"The sum of the 10 term of the Fibonacci is 143."	"The sum of the 10 term of the Fibonacci is 143."
20	"The sum of the 20 term of the Fibonacci is 17710.	"The sum of the 20 term of the Fibonacci is 17710.

```c
#include <stdio.h>

int fib(int n);
int sum(int n);

int main() {
    int n;
    printf("Please enter the nth term.\n");
    scanf("%d",&n);
    printf("The sum of the %d term of the Fibonacci is %d.\n",n,sum(n));
}
//Sums the values of the Fibonacci Sequence up to the nth term.
int sum(int n){
    int sum = 0;
    for(int i=1; i <= n; i++) {
        sum += fib(i);
    }
    return sum;
}

//Function that calculates the value of a fibonacci term.
int fib(int n) {
    //Stopping condition - the first and second Fibonacci Terms are both 1.
    if(n==1 || n==2){
        return 1;
    } else {
        return fib(n-1)+fib(n-2);
    }
}
```