Array Data Structure

1A. Aim: Write and implement a program to reverse the given array.

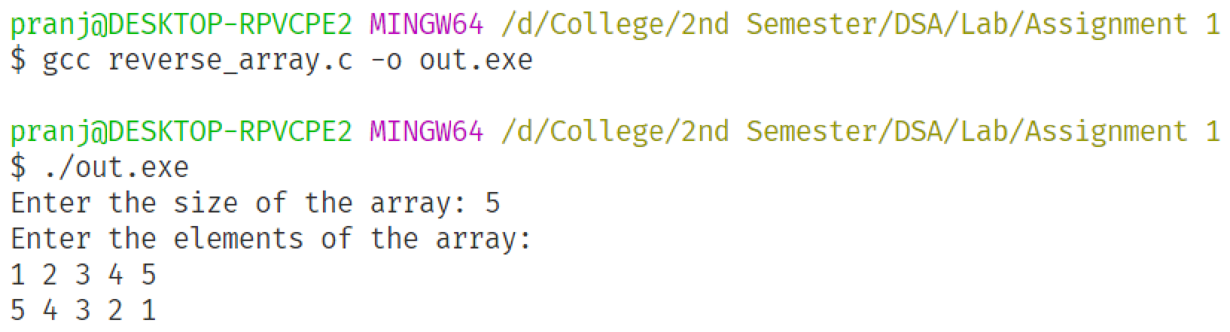
Algorithm:

Step 1: Initialize variable to store size of the array and input the size of the array.
Step 2: Allocate memory from the heap for the array.
Step 3: Use a loop to input the elements of the array from the user.
Step 4: Allocate memory from the heap for the reversed.
Step 5: Use a for loop with counter i from 0 to size – 1, and put reversed [i] = array[size-i-1].
Step 6: Use a for loop do display the contents of the reversed array.

Program: [In next page]

```c
// Exp 1A: Aim: Reverse the given array;
// Author: Pranjal Timsina, 20BDS0392
#include <stdio.h>
#include <stdlib.h>
void print_array(int *array, int size){
    // Iterate through the elements of the array and prints it
    for (int i = 0; i < size; i++){
        printf("%d ", array[i]);
    }
    printf("\n");
}
void reverse(int *array,int* reversed_array, int size){
     // First element of reversed is the element
     // size-i-0 of the main array
     // Similarly the second element of reversed is the element
     // size-i-1 of the main array
    for (int i = 0; i < size; i++){
        reversed_array[i] = array[size-i-1];
    }
}
int main() {
    int size; // Initialize variable to store size of the array
    printf("Enter the size of the array: "); // Message to user
    scanf("%d", &size); // Read the size from user input
    // Allocate memory for the array from heap
    int *array = calloc(size, sizeof(int));
    // Input elements of the array
    printf("Enter the elements of the array:\n");
    for (int i = 0; i < size; i++){
        scanf("%d", &array[i]);
    }
    //Allocate memory for the reversed array from the heap
    int *reversed_array = calloc(size, sizeof(int));
    // Function call to reverse array
    reverse(array, reversed_array, size);
    // Function call to print the reversed array
    print_array(reversed_array, size);
    // Free the memory used by the array and reversed array
    free(array);
    free(reversed_array);
    return 0;
}
```

Output:



```
pranj@DESKTOP-RPVCPE2 MINGW64 /d/College/2nd Semester/DSA/Lab/Assignment 1
$ gcc reverse_array.c -o out.exe

pranj@DESKTOP-RPVCPE2 MINGW64 /d/College/2nd Semester/DSA/Lab/Assignment 1
$ ./out.exe
Enter the size of the array: 5
Enter the elements of the array:
1 2 3 4 5
5 4 3 2 1
```

Results:

Thus, the program to reverse the given array is implemented with time complexity $O(n)$.

1B. Aim: Write a program to rotate the elements of an array by d elements.

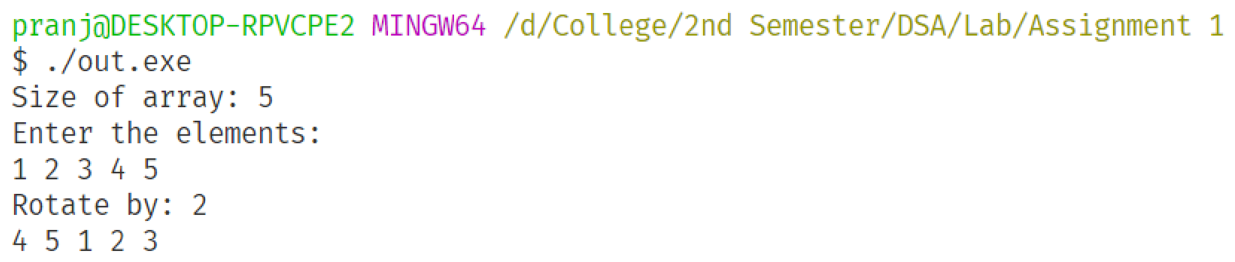
Algorithm:

Step 1: Initialize variable to store size of the array and input the size of the array.
Step 2: Allocate memory from the heap for the array.
Step 3: Use a loop to input the elements of the array from the user
Step 4: Initialize a variable to store the degree of rotation and input the degree from user.
Step 5: Allocate memory from the heap for the rotated array
Step 6: Use a for loop to iterate through the elements of the array and insert the i^{th} item in the new array's index $(i + degree)\ \%\ size$.

Program: [In next page]

```c
// Exp 1B; Aim: Rotate the array by d elements
// Author: Pranjal Timsina, 20BDS0392
#include <stdio.h>
#include <stdlib.h>

int main() {
    int size;
    printf("Size of array: ");
    scanf("%d", &size);

    // Allocate memory from the heap to the array
    int *arr = calloc(size, sizeof(int));

    // Input elements of the array
    printf("Enter the elements:\n");
    for (int i = 0; i < size; i++){
        scanf("%d", &arr[i]);
    }

    //Input the degree to rotate by
    int degree;
    printf("\nRotate by: ");
    scanf("%d", &degree);

    // Allocate memory from heap to rotated array
    int *new = calloc(size, sizeof(int));
    int temp; // A temporary variable

    // New index is current_index+degree, but
    // since current_index+degree can go beyond the size of array
    // (current_index + degree) % size is used
    for (int i = 0; i <size; i++){
        temp = (i+degree) % size;
        new[temp] = arr[i];
    }
    // Print the elements of the rotated array
    for (int i = 0; i < size; i++){
        printf("%d ", new[i]);
    }
    // Free the allocated memory from the heap
    free(arr);
    free(new);
}
```

Output:



```
pranj@DESKTOP-RPVCPE2 MINGW64 /d/College/2nd Semester/DSA/Lab/Assignment 1
$ ./out.exe
Size of array: 5
Enter the elements:
1 2 3 4 5
Rotate by: 2
4 5 1 2 3
```

Results:

Thus, the program to rotate the given array is implemented with time complexity $O(n)$.

1C. Aim: Given two strings S_1 and S_2, find if S_1 is a substring of S_2. If yes, return the index of the first occurrence, else return -1.

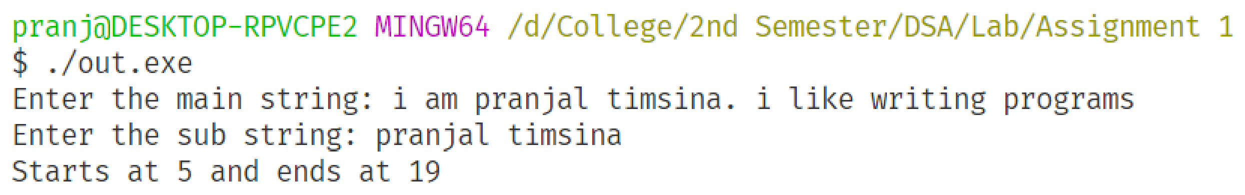
Algorithm:

Step 1: Initialize variable for two char arrays and input the S_1 and S_2.
Step 2: Traverse through each of the arrays and calculate the length of S_1 and S_2.
Step 3: Traverse through the main string and find the index where the character is the same as the first character of the sub string.
Step 4: Check the next n characters of the main string to check if it matches the substring, where n is the length of the sub string.
Step 5: If the substring is found, print the index and exit the program.
Step 6: If the loop iterates completely without match, return -1.

Program: [In next page]

```
// Exp 1C; Aim: Print the index of first occurrent of substring
// Author: Pranjal Timsina, 20BDS0392
#include <stdio.h>
#include <stdlib.h>

int get_length(char* string){
    // Function to get the length of a string;
    // Traverses the char array until a \0 is found
    int size = 0;
    for (int i = 0; i < 499;i++){
        if (string[i] == '\0'){
            break;
        }
        size++;
    }
    return size;
}

int match(char main_string[500], char sub_string[500], int i, int s
ub_len){
// function to match if a sub string of length sub_len starts at
// index i of main_string
    for (int counter = i; counter < i+sub_len; counter++){
        if (main_string[counter] != sub_string[counter-i]){
            return 0;
        }
    }
    return 1;
}

int main() {

    printf("Enter the main string: ");
    char main_string[500];
    scanf("%[^\n]s", &main_string);
    getchar();

    printf("Enter the sub string: ");
    char sub_string[500];
    scanf("%[^\n]s", &sub_string);

    int main_length = get_length(main_string);
    int sub_length = get_length(sub_string);
    if (sub_length > main_length){
        // Edge case handling
        printf("Error!
                Substring cannot be longer than the main string!");
        return 0;
    } else{
        // traverses the main char array and checks if the current
       // character matches
        // with the first character of the sub_string
        for (int i = 0; i < main_length-sub_length+1; i++){
            if (main_string[i] == sub_string[0]){
                if (match(main_string, sub_string,i, sub_length)) {
                    // checks if substring exists, starting at posi
                   // tion i
                    printf("Starts at %d and ends at %d",
                           i, i+sub_length-1);
                    break;
                }
            }
        }
    }

    return 0;
}
```

Output:



```
pranj@DESKTOP-RPVCPE2 MINGW64 /d/College/2nd Semester/DSA/Lab/Assignment 1
$ ./out.exe
Enter the main string: i am pranjal timsina. i like writing programs
Enter the sub string: pranjal timsina
Starts at 5 and ends at 19
```

Results:

Thus, the program to find the index of the substring is implemented with time complexity $O(nm)$. Where n is the length of the main string and m is the length of the substring.

1D. Aim: Write a program to insert an element in an array at a specified location.

Algorithm:

Step 1: Input size of array from the user and allocate memory for the array
Step 2: Create another array of length size to store whether a particular index in an array is
    occupied or not.
Step 3: Take input from the user for the position and data
Step 4: If the position is within bounds and the index is unoccupied, insert data in the position
Step 5: Try to shift the contents of the array cyclically right starting from the position till a
    0 is encountered.
Step 6: If shifting is successful, insert data in the position
Step 7: If shifting is unsuccessful due to array being full, ask the user whether they want to
    increase the size of the array.
Step 8: If the user chooses to increase the size of the array then shift data cyclically and insert
    data in the position. Then go to step 3.
Step 9: If the user chooses not to increase the size of the array, exit.

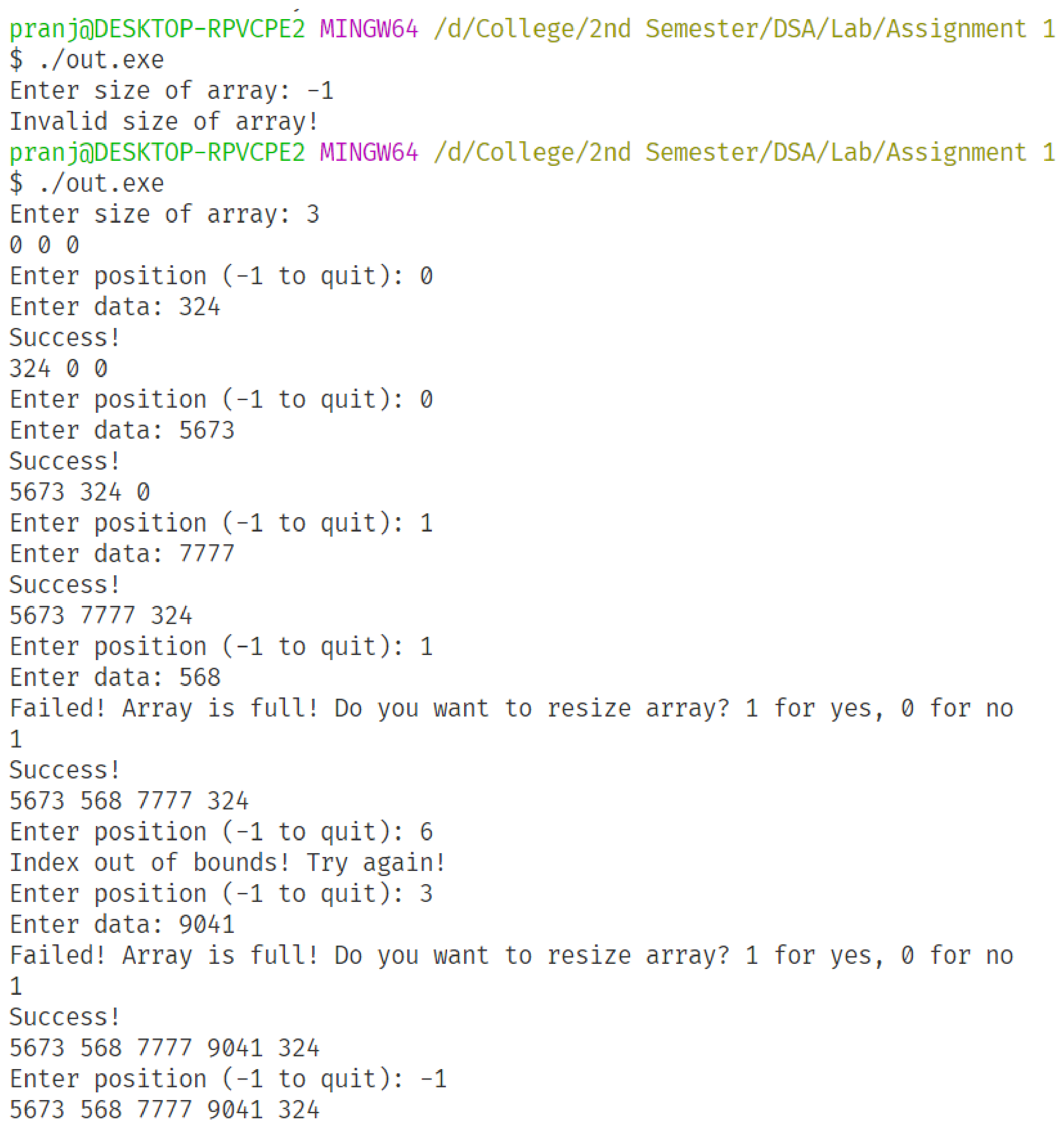
Program: [In next page]

```c
// Exp 1D; Aim: Insert element at a given position in an array
// Author: Pranjal Timsina; 20BDS0392
#include <stdio.h>
#include <stdlib.h>
void print_array(int *array, int size) {
    // function to print array
    for (int i = 0; i < size; i++){
        printf("%d ", array[i]);
    }
    printf("\n");
}
int insert(int *array, int *occupied,int size,
           int position, int data)
{
    // Start at the position, shift everything cyclically right
    //until a 0 is found
    int found = -1; // initally no empty location is found
    for (int i = position+1;i != position; i = (i+1)%size){
        // loop to check for the first unoccupied
        // location in the array
        if (occupied[i] == 0){
            found = i;
            break;
        }
    }
    if (found == -1){
        // if there are no unoccupied locations return 0
        // indicating a full array
        return 0;
    } else {
        // start at the empty location and shift all the
        // elements to make space in position
        int current = found;
        int previous;
        while (1) {
            // current index of array takes element from the
            // index left of it
            previous = current - 1;
            // condition to make shifting cyclic
            if (previous < 0) previous = size+previous;
            // swapping elements
            array[current] = array[previous];
            occupied[current] = 1;
```

```
                    current = previous;
                // if it reaches the required position, breaks
                if (previous == position) break;
            }
        }
    array[position] = data; // inserting data
    return 1; // returning success code
}

int main() {
    // taking user input for size of array
    int size;
    printf("Enter size of array: ");
    scanf("%d", &size);

    // validating input size
    if (size <= 0) {
        printf("Invalid size of array!");
        return -1;
    }

    // allocating memory for the array
    int *array = calloc(size, sizeof(int));
    // allocating memory for array to store whether an index is
    // occupied or not
    int *occupied = calloc(size, sizeof(int));

    // checking for memory allocation failure
    if (array == NULL || occupied == NULL) {
        printf("Could not allocate memory!");
        return -1;
    }

    // variables for index, data
    // temp used for indicating whether to expand array
    int position, data, temp;
    while(1) {
        printf("Enter position (-1 to quit): ");
        scanf("%d", &position);
        // exit condition
        if (position == -1){
            print_array(array, size);
            free(array);
            free(occupied);
            return 0;
        } else if(position >= size || position < -1) {
            // index validation
            printf("Index out of bounds! Try again!\n");
            continue;
        } else {
            printf("Enter data: ");
            scanf("%d", &data);
            if (!occupied[position]) {
                // if unoccupied, directly store data
                array[position] = data;
                occupied[position] = 1;
                printf("Success!\n");
            } else {
                // calls insert function which returns true if
               // success
                // false if array is full
                if (insert(array, occupied,size, position, data)) {
                    printf("Success!\n");
                } else {
                    // asking user to resize array
                     printf("Failed! Ar-
ray is full! Do you want to resize array? 1 for yes, 0 for no:\n");
                    scanf("%d", &temp);
                    if (temp){
                        // expanding size of array
                        array = realloc(array, size+1);
                        occupied = realloc(occupied, size+1);
                        size++; // incrementing the size of array
                        occupied[size-1] = 0;
               // changing garbage value to false for new index
                        // validating error in memory allocation
                        if (array == NULL || occupied == NULL) {
                            printf("Could not allocate memory!");
                            free(array);
                            free(occupied);
                            return 0;
                        }
                        // inserting data in array
                        if (insert(array, occupied, size, posi-
tion, data)){
```

```
                        printf("Success!\n");

                            } else {
                                printf("Failure!\n");
                            }
                        } else {
                            // exits if user chooses not to expand full
 array
                            printf("Exiting the program!");
                            return 0;
                        }
                    }
                }
            }
            print_array(array, size);
        }
    // deallocating memory
    free(array);
    free(occupied);
    return 0;
}
```

Output: [In next page]



```
pranj@DESKTOP-RPVCPE2 MINGW64 /d/College/2nd Semester/DSA/Lab/Assignment 1
$ ./out.exe
Enter size of array: -1
Invalid size of array!
pranj@DESKTOP-RPVCPE2 MINGW64 /d/College/2nd Semester/DSA/Lab/Assignment 1
$ ./out.exe
Enter size of array: 3
0 0 0
Enter position (-1 to quit): 0
Enter data: 324
Success!
324 0 0
Enter position (-1 to quit): 0
Enter data: 5673
Success!
5673 324 0
Enter position (-1 to quit): 1
Enter data: 7777
Success!
5673 7777 324
Enter position (-1 to quit): 1
Enter data: 568
Failed! Array is full! Do you want to resize array? 1 for yes, 0 for no
1
Success!
5673 568 7777 324
Enter position (-1 to quit): 6
Index out of bounds! Try again!
Enter position (-1 to quit): 3
Enter data: 9041
Failed! Array is full! Do you want to resize array? 1 for yes, 0 for no
1
Success!
5673 568 7777 9041 324
Enter position (-1 to quit): -1
5673 568 7777 9041 324
```

Results:

Thus, the program to insert and element in an array at a specified location was implemented.

1E. Aim: Write a program to find the median of two arrays of the same size. (Use bubble sort)

Algorithm:

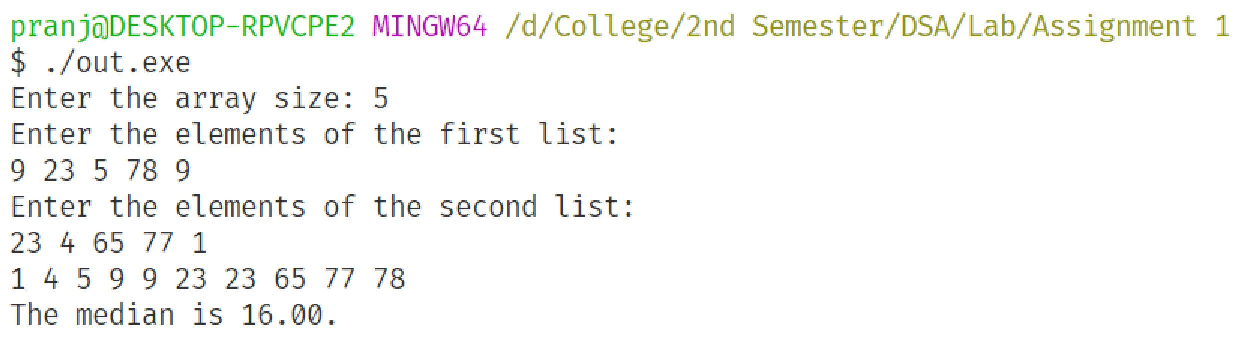
Step 1: Initialize variable for the size of the arrays.
Step 2: Allocate memory for the two arrays from the heap. Allocate twice the size for the first array to easily merge the two arrays later.
Step 3: Append the elements of the second array to the end of the first array.
Step 4: Bubble sort the merged array.
Step 5: Calculate the median which will be the average of merged_list[size] and merged_list[size-1] as the number of elements in the merged_list is always even.
Step 6: Display the median of the merged array.

Program: [In next page]

```
// Exp 1E; Aim: Print the median of two arrays
// Author: Pranjal Timsina, 20BDS0392
#include <stdio.h>
#include <stdlib.h>
void print_array(int *array, int length) {
    // function to iterate through the array and print its content
    for (int i = 0; i < length; i++) {
        printf("%d ", array[i]);
    }
    printf("\n");
}
void bubble_sort(int *array, int length){
    // function to implement bubble sort
    int current, next, temp;
    for (int i = 0; i < length; i++) {
        for (int j = 0; j < length-i-1; j++) {
                if (array[j] > array[j+1]) {
                    temp = array[j+1];
                    array[j+1] = array[j];
                    array[j] = temp;
                }
        }
    }
}

int main() {
    int size;
    printf("Enter the array size: ");
    scanf("%d", &size);

    // allocating memory for the two arrays
    // list_a is given twice the size so that list_b can be merged
    // later
    int *list_a = calloc(size*2, sizeof(int));
    int *list_b = calloc(size, sizeof(int));

    // input elements
    printf("Enter the elements of the first array:\n");
    for (int i = 0; i < size; i++) {
        scanf("%d", &list_a[i]);
    }
    printf("Enter the elements of the second array:\n");
    for (int i = 0; i < size; i++) {
```

```c
        scanf("%d", &list_b[i]);
    }

    // merge the two lists
    for (int i = size; i < size*2; i++){
        list_a[i] = list_b[i-size];
    }

    bubble_sort(list_a, size*2); // call function to sort array
    print_array(list_a, size*2); // call function to print array

    // since the size of the merged array is 2*size,
    // the number of items will always be even

    // logic to calculate the median and print it
    double median = ((double) list_a[size-1]
                    + (double) list_a[size]) / 2.0;
    printf("The median is %.2lf.", median);

    // free allocated memory
    free(list_a);
    free(list_b);
    return 0;
}
```

Output:



```
pranj@DESKTOP-RPVCPE2 MINGW64 /d/College/2nd Semester/DSA/Lab/Assignment 1
$ ./out.exe
Enter the array size: 5
Enter the elements of the first list:
9 23 5 78 9
Enter the elements of the second list:
23 4 65 77 1
1 4 5 9 9 23 23 65 77 78
The median is 16.00.
```

Results:

Thus, the program to find the median of two lists was implemented.

1F. Aim: Write a program to print all the repeated numbers in an array with frequency.

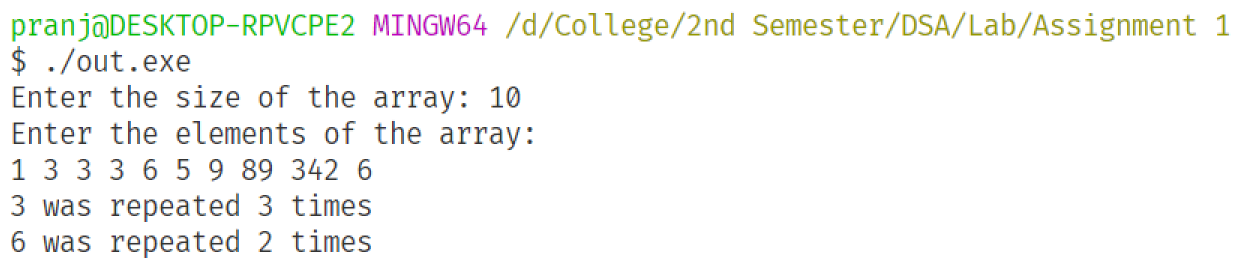
Algorithm:

```
Step 1: Initialize the variable for the size of array and get user input
Step 2: Allocate memory from the heap for the array
Step 3: Sort the array using insertion sort, or a more efficient sorting algorithm so that the
        repeated numbers will be adjacent to each other in the array.
Step 4: Initialize a counter variable to 1
Step 5: Traverse the array and compare with next.
Step 6: If the next item is different and check if counter > 1 and then print the
        Frequency, then reset counter to 1.
Step 7: If the next item is the same increment the counter variable.
Step 8: After the array is fully traversed exit the program
```

Program: [In next page]

```c
// Exp 1F; Aim: Print the frequency of repeated elements in array
// Author: Pranjal Timsina, 20BDS0392
#include <stdio.h>
#include <stdlib.h>

// function to sort array, so repeated elements are grouped
// together
void sort(int * array, int size) {
    int temp; // temporary variable to swap array elements

    // Logic for insertion sort
    for (int i = 1; i < size; i++){
        for (int j = 0; j < i;j++){
            if (array[i] < array[j]){
                temp = array[j];
                array[j] = array[i];
                array[i] = temp;
            }
        }
    }
}

int main() {

    // Declare and input the size of the array
    int size;
    printf("Enter the size of the array: ");
    scanf("%d", &size);

    // Allocate memory to the array based on the given size
    int *array = calloc(size, sizeof(int));

    // Input the elements of the array
    printf("Enter the elements of the array:\n");
    for (int i = 0; i < size; i++) {
        scanf("%d", &array[i]);
    }

    sort(array, size); // Calls the function to sort the given
                       //array

    int repeated = 1; // Each number exist at least once in the
                      // array
    // Variables to check for adjacent elements
    int previous = array[0];
    int current;

    // Since the array is sorted, repeated elements will be grouped
    // together
    // This loop looks at the number of times current and previous
    //are the same,
    // and stores it in the variable repeated

    for (int i = 1; i < size; i++) {
        current = array[i];
        if (previous == current) {
            repeated++;
            continue;
        } else {
            // Control flow reaches here if previous and current
           // elements are different
            if (repeated != 1) {
                // If more than one instance of an elmement exists,
               // then prints the element's count
                printf("%d was repeated %d times\n", previous,
                repeated);
            }
            // Resets the repeated count and sets the previous valu
e
            repeated = 1;
            previous = current;
        }
    }

    // Edge case handling for the last element in the sorted array
    if (repeated > 1) {
        printf("%d was repeated %d times\n", current, repeated);
    }

    free(array); // Frees the memory allocated to the array
}
```

<u>Output:</u>



```
pranj@DESKTOP-RPVCPE2 MINGW64 /d/College/2nd Semester/DSA/Lab/Assignment 1
$ ./out.exe
Enter the size of the array: 10
Enter the elements of the array:
1 3 3 3 6 5 9 89 342 6
3 was repeated 3 times
6 was repeated 2 times
```

<u>Results:</u>

Thus, the program to print all the repeated numbers in an array with frequency was implemented.

1G. Aim: Write a program to multiply two matrices.

Algorithm:

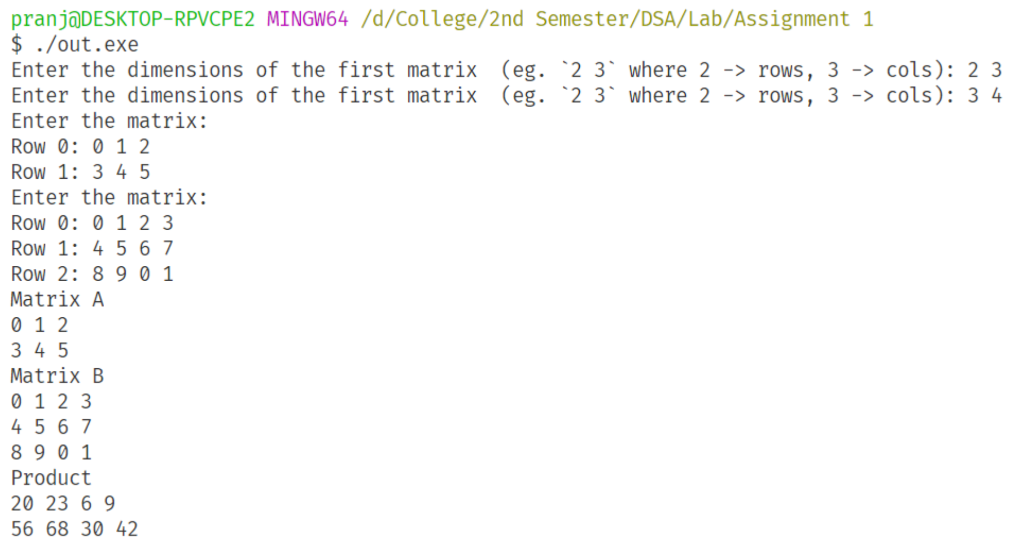
```
Step 1: Input the dimensions for the two matrices
Step 2: Check if the dimensions are valid for multiplication
Step 3: If the dimensions are invalid, exit with error message
Step 4: Input the elements of the two matrices
Step 5: Initialize a 2D product array with rows $R_1$ and columns $C_2$.
Step 6: Iterate through the indices of the product array.
Step 7: The value of the element[i][j] of the product array is given by:
        element[i][j] = a[i][h]*b[h][j], where h = 0 to $C_1$.
Step 8: Display the product.
```

Program: [In next page]

```
// Exp 1G; Aim: Multiply the given two matrices
// Author: Pranjal Timsina; 20BDS0392
#include <stdio.h>
#include <stdlib.h>
void print_matrix(int rows, int cols, int matrix[][cols]) {
    // function to print the matrix
    for (int i = 0; i < rows; i++){
        for(int j = 0; j < cols; j++){
            printf("%d ", matrix[i][j]);
        }
        printf("\n");
    }
}
void input_matrix(int rows, int cols, int matrix[][cols]){
    // function to input a matrix
    printf("Enter the matrix:\n");
    for (int i = 0; i < rows; i ++){
        printf("Row %d: ", i);
        for (int j = 0; j < cols; j++){
            scanf(("%d"), &matrix[i][j]);
        }
    }
}

int main() {
    // Initializing the dimensions of the matrices and
    // taking user input
    int row1, row2, col1, col2;
    printf("Enter the dimensions of the first ma-
trix  (eg. `2 3` where 2 -> rows, 3 -> cols): ");
    scanf("%d%d", &row1, &col1);
    printf("Enter the dimensions of the first ma-
trix  (eg. `2 3` where 2 -> rows, 3 -> cols): ");
    scanf("%d%d", &row2, &col2);

    // Error handling for invalid dimensions
    if (col1 != row2) {
        printf("Invalid dimensions.");
        return 0;
    }
    // Declaring the 2D arrays for matrices
    int matrix_a[row1][col1];
    int matrix_b[row2][col2];
```

```c
    int product[row1][col2];
    // calling the input functions
    input_matrix(row1, col1, matrix_a);
    input_matrix(row2, col2, matrix_b);
    // product[i][j] = A[i][1]B[1][j] + A[i][2]B[2][j]...
    int temp;
    for (int i = 0; i < row1; i++){
        for (int j = 0; j < col2; j++) {
            int temp = 0;
            for (int h =0; h < row2;h++){
                temp+= matrix_a[i][h]*matrix_b[h][j];
            }
            product[i][j] = temp;
        }
    }

    printf("Matrix A \n");
    print_matrix(row1, col1, matrix_a);
    printf("Matrix B \n");
    print_matrix(row2, col2, matrix_b);

    printf("Product \n");
    print_matrix(row1, col2, product);
}
```

<u>Output:</u>



```
pranj@DESKTOP-RPVCPE2 MINGW64 /d/College/2nd Semester/DSA/Lab/Assignment 1
$ ./out.exe
Enter the dimensions of the first matrix  (eg. `2 3` where 2 -> rows, 3 -> cols): 2 3
Enter the dimensions of the first matrix  (eg. `2 3` where 2 -> rows, 3 -> cols): 3 4
Enter the matrix:
Row 0: 0 1 2
Row 1: 3 4 5
Enter the matrix:
Row 0: 0 1 2 3
Row 1: 4 5 6 7
Row 2: 8 9 0 1
Matrix A
0 1 2
3 4 5
Matrix B
0 1 2 3
4 5 6 7
8 9 0 1
Product
20 23 6 9
56 68 30 42
```

<u>Results:</u>

Thus, the program multiply two matrices was implemented.