1. Write a C program that dynamically allocates memory for a 2D array of integers with dimensions specified by the user. The program should implement a custom sorting algorithm that sorts the rows of the array in descending order based on the product of the elements in each row. If two rows have the same product, they should appear in the same order as the input. Additionally, sort the columns in ascending order based on the number of odd numbers in each column. Implement separate functions for sorting rows and columns to keep the program modular.

   Ensure proper error handling for memory allocation failures by displaying appropriate error messages and terminating the program if needed (you may use exit(0) to exit the program prematurely, and remember to #include <stdlib.h>). The sorting algorithm and function design are up to you, and you may use a temporary dynamically allocated 1D array if required. Make sure to free all dynamically allocated memory at the end of the program.

   The program should take the number of rows and columns as input from the user and dynamically allocate the 2D array based on these dimensions.

   Example 1:

   Enter the number of rows: 3

   Enter the number of columns: 3

   Enter the elements of the array:

   2 4 6

   1 3 5

   7 9 11

   Array after sorting rows by product and columns by odd count:

   7 9 11

   2 4 6

   1 3 5

   Example 2:

   Enter the number of rows: 4

   Enter the number of columns: 2

   Enter the elements of the array:

   2 8

3 9

4 6

1 2

Array after sorting rows by product and columns by odd count:

9 3

6 4

8 2

2 1

```
/*1.    Write a C program that dynamically allocates memory for a 2D array
of integers with dimensions specified by the user. The program should
implement a custom sorting algorithm that sorts the rows of the array in
descending order based on the product of the elements in each row. If two
rows have the same product, they should appear in the same order as the
input. Additionally, sort the columns in ascending order based on the
number of odd numbers in each column. Implement separate functions for
sorting rows and columns to keep the program modular.
Ensure proper error handling for memory allocation failures by displaying
appropriate error messages and terminating the program if needed (you may
use exit(0) to exit the program prematurely, and remember to #include
<stdlib.h>). The sorting algorithm and function design are up to you, and
you may use a temporary dynamically allocated 1D array if required. Make
sure to free all dynamically allocated memory at the end of the program.
The program should take the number of rows and columns as input from the
user and dynamically allocate the 2D array based on these dimensions.
Example 1:
Enter the number of rows: 3
Enter the number of columns: 3
Enter the elements of the array:
2 4 6
1 3 5
7 9 11
Array after sorting rows by product and columns by odd count:
7 9 11
2 4 6
1 3 5
Example 2:
Enter the number of rows: 4
Enter the number of columns: 2
Enter the elements of the array:
2 8
3 9
4 6
1 2
Array after sorting rows by product and columns by odd count:
9 3
6 4
8 2
2 1
*/

// Code creator: Nishkal Prakash (nishkal@iitkgp.ac.in)
// Date: 2024/10/22
```

```c
// Program to sort rows in descending order based on product of elements
and columns in ascending order based on number of odd numbers
#include <stdio.h>
#include <stdlib.h>

// Function to sort rows in descending order based on product of elements
void sortRows(int **arr, int rows, int cols)
{
    int *product = (int *)malloc(rows * sizeof(int));
    int *temp;
    int tempProduct, i, j;

    if (product == NULL)
    {
        printf("Memory allocation failed. Exiting program.\n");
        exit(0);
    }

    // Calculate product of each row
    for (i = 0; i < rows; i++)
    {
        product[i] = 1;
        for (j = 0; j < cols; j++)
        {
            product[i] *= arr[i][j];
        }
    }

    // Custom sorting algorithm
    for (i = 0; i < rows - 1; i++)
    {
        for (j = 0; j < rows - i - 1; j++)
        {
            if (product[j] < product[j + 1])
            {
                // Swap rows
                temp = arr[j];
                arr[j] = arr[j + 1];
                arr[j + 1] = temp;

                // Swap product values
                tempProduct = product[j];
                product[j] = product[j + 1];
                product[j + 1] = tempProduct;
            }
```

```
        }
    }

    free(product);
}

// Function to sort columns in ascending order based on number of odd
numbers
void sortColumns(int **arr, int rows, int cols)
{
    int *oddCount = (int *)malloc(cols * sizeof(int));
    int i, j, temp, k, tempCount;
    if (oddCount == NULL)
    {
        printf("Memory allocation failed. Exiting program.\n");
        exit(0);
    }

    // Calculate number of odd numbers in each column
    for (j = 0; j < cols; j++)
    {
        oddCount[j] = 0;
        for (i = 0; i < rows; i++)
        {
            if (arr[i][j] % 2 != 0)
            {
                oddCount[j]++;
            }
        }
    }

    // Custom sorting algorithm
    for (j = 0; j < cols - 1; j++)
    {
        for (k = 0; k < cols - j - 1; k++)
        {
            if (oddCount[k] > oddCount[k + 1])
            {
                // Swap columns
                for (i = 0; i < rows; i++)
                {
                    temp = arr[i][k];
                    arr[i][k] = arr[i][k + 1];
                    arr[i][k + 1] = temp;
                }
```

```
                // Swap odd count values
                tempCount = oddCount[k];
                oddCount[k] = oddCount[k + 1];
                oddCount[k + 1] = tempCount;
            }
        }
    }

    free(oddCount);
}

int main()
{
    int rows, cols;

    // Input number of rows and columns
    printf("Enter the number of rows: ");
    scanf("%d", &rows);
    printf("Enter the number of columns: ");
    scanf("%d", &cols);

    // Dynamically allocate memory for 2D array
    int **arr = (int **)malloc(rows * sizeof(int *));
    if (arr == NULL)
    {
        printf("Memory allocation failed. Exiting program.\n");
        exit(0);
    }

    for (int i = 0; i < rows; i++)
    {
        arr[i] = (int *)malloc(cols * sizeof(int));
        if (arr[i] == NULL)
        {
            printf("Memory allocation failed. Exiting program.\n");
            exit(0);
        }
    }

    // Input elements of the array
    printf("Enter the elements of the array:\n");
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
```

```
        {
            scanf("%d", &arr[i][j]);
        }
    }

    // Sort rows based on product of elements
    sortRows(arr, rows, cols);

    // Sort columns based on number of odd numbers
    sortColumns(arr, rows, cols);

    // Display the sorted array
    printf("Array after sorting rows by product and columns by odd
count:\n");
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < cols; j++)
        {
            printf("%d ", arr[i][j]);
        }
        printf("\n");
    }

    // Free dynamically allocated memory
    for (int i = 0; i < rows; i++)
    {
        free(arr[i]);
    }
    free(arr);

    return 0;
}
```

2. Write a C program that takes a string as input and performs a two-step transformation:
    1. Step 1: Remove all consecutive duplicate characters, leaving only one occurrence of each character.
    2. Step 2: For the modified string from Step 1, group the characters by whether they are vowels or consonants, maintaining the order in which they appeared. The program should then return the modified string in the format Vowels: <vowel_string> Consonants: <consonant_string>. If there are no vowels or consonants, that part of the output should be omitted.

    If the original string does not contain any consecutive duplicates or if it contains only vowels or consonants, the program should still perform both steps and format the output as described.

    Example 1:
    Enter a string: AAAaaBBBbbbAaCc
    String after deleting the consecutive duplicate characters: AaBbAaCc
    Vowels: AaAa Consonants: BbCc

    Example 2:
    Enter a string: xyz
    String after deleting the consecutive duplicate characters: xyz
    Consonants: xyz

    Example 3:
    Enter a string:
    String after deleting the consecutive duplicate characters:
    The input string is empty.

```c
/*
2.  Write a C program that takes a string as input and performs a two-step transformation:
Step 1: Remove all consecutive duplicate characters, leaving only one occurrence of each character.
Step 2: For the modified string from Step 1, group the characters by whether they are vowels or consonants, maintaining the order in which they appeared. The program should then return the modified string in the format Vowels: <vowel_string> Consonants: <consonant_string>. If there are no vowels or consonants, that part of the output should be omitted.
If the original string does not contain any consecutive duplicates or if it contains only vowels or consonants, the program should still perform both steps and format the output as described.
Example 1:
Enter a string: AAAaaBBBbbbAaCc
String after deleting the consecutive duplicate characters: AaBbAaCc
Vowels: AaAa Consonants: BbCc

Example 2:
Enter a string: xyz
String after deleting the consecutive duplicate characters: xyz
Consonants: xyz

Example 3:
Enter a string:
String after deleting the consecutive duplicate characters:
The input string is empty.
*/

// Code creator: Nishkal Prakash (nishkal@iitkgp.ac.in)
// Date: 2024/10/22
// Program to remove consecutive duplicate characters and group characters by vowels and consonants

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

// Function to check if a character is a vowel
int isVowel(char c)
{
    c = tolower(c);
    return (c == 'a' || c == 'e' || c == 'i' || c == 'o' || c == 'u');
}
```

```c
// Function to remove consecutive duplicate characters
void removeConsecutiveDuplicates(char *str)
{
    if (str[0] == '\0')
    {
        printf("String after deleting the consecutive duplicate
characters:\n");
        return;
    }

    int len = strlen(str);
    int index = 0;

    for (int i = 0; i < len; i++)
    {
        if (str[i] != str[i + 1])
        {
            str[index++] = str[i];
        }
    }

    str[index] = '\0';
    printf("String after deleting the consecutive duplicate characters:
%s\n", str);
}

// Function to group characters by vowels and consonants
void groupCharacters(char *str)
{
    int len = strlen(str);
    char vowels[len], consonants[len];
    int vIndex = 0, cIndex = 0;

    for (int i = 0; i < len; i++)
    {
        if (isVowel(str[i]))
        {
            vowels[vIndex++] = str[i];
        }
        else
        {
            consonants[cIndex++] = str[i];
        }
    }
```

```
    vowels[vIndex] = '\0';
    consonants[cIndex] = '\0';

    if (vIndex > 0)
    {
        printf("Vowels: %s ", vowels);
    }

    if (cIndex > 0)
    {
        printf("Consonants: %s\n", consonants);
    }
}

int main()
{
    char str[100];

    printf("Enter a string: ");
    fgets(str, sizeof(str), stdin);

    if (str[0] == '\n')
    {
        printf("The input string is empty.\n");
        return 0;
    }

    str[strcspn(str, "\n")] = '\0';

    removeConsecutiveDuplicates(str);
    groupCharacters(str);

    return 0;
}
```

3. Write a C program that takes a string as input and implements two functions: one function that removes all occurrences of a specified character from the string and another function that counts the total number of substrings in the modified string, treating all substrings as distinct, even if they consist of the same characters in different positions.

A substring should be considered distinct based on its position in the original modified string, even if it appears multiple times.

Example 1:

Enter a string: banana
Enter the character to remove: a

The modified string after removing 'a' is: bnn
Number of distinct substrings in bnn: 6

Explanation: The distinct substrings are "b", "n", "nn", "bn", "bnn", and "n".

```
/*
Write a C program that takes a string as input and implements two
functions: one function that removes all occurrences of a specified
character from the string and another function that counts the total
number of substrings in the modified string, treating all substrings as
distinct, even if they consist of the same characters in different
positions.

A substring should be considered distinct based on its position in the
original modified string, even if it appears multiple times.

Example 1:
Enter a string: banana
Enter the character to remove: a

The modified string after removing 'a' is: bnn
Number of distinct substrings in bnn: 6
Explanation: The distinct substrings are "b", "n", "nn", "bn", "bnn", and
"n".

*/

// Code creator: Nishkal Prakash (nishkal@iitkgp.ac.in)
// Date: 2024/10/22
// Program to remove all occurrences of a specified character and count
the total number of distinct substrings
```

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

// Function to remove all occurrences of a specified character from the
string
void removeCharacter(char *str, char ch)
{
    int len = strlen(str);
    int index = 0;

    for (int i = 0; i < len; i++)
    {
        if (str[i] != ch)
        {
            str[index++] = str[i];
        }
    }

    str[index] = '\0';
    printf("The modified string after removing '%c' is: %s\n", ch, str);
}

// Function to count the total number of distinct substrings
void countDistinctSubstrings(char *str)
{
    int len = strlen(str);
    int count = 0;

    for (int i = 0; i < len; i++)
    {
        for (int j = i; j < len; j++)
        {
            count++;
        }
    }

    printf("Number of distinct substrings in %s: %d\n", str, count);
}

int main()
{
    char str[100], ch;

    // Input string and character to remove
```

```
    printf("Enter a string: ");
    scanf("%s", str);
    printf("Enter the character to remove: ");
    scanf(" %c", &ch);

    removeCharacter(str, ch);
    countDistinctSubstrings(str);

    return 0;
}
```