In this Lab Exam, you are going to solve 5 questions that test your general knowledge about the topics we have covered in Ceng140

You will implement 5 functions that are:

- **CalculatePos**
- **TransposeSquareMatrix**
- **ProcessShipment**
- **ReverseLinkedList**
- **SplitCircularLLs**

**Struct Descriptions:**

1. StationNode:
    - The **StationNode** struct represents a linked list node
        - **id** (integer) : id of the station node
        - **float** (coordX): x-coord of the station node
        - **float** (coordY): y-coord of the station node
        - **next** (pointer to StationNode): a pointer to the next node in the list
2. Response:
    - The **Response** struct is a struct to be filled inside the **ProcessShipment** function (it is the same as you encountered in LabExam 4 Preliminary)
        - **username** (char array): the username of the client
        - **data** (union): data is a union which holds two struct instances called **error** and **success**
            - **Error** (struct):
                - **code** (char): error code is held as a char
            - **Success** (struct):
                - **totalDistance** (float): total distance calculated between the stations is stored here
                - **stationCount** (int): the number of station nodes counted in the list is stored here
3. ListNode:
    - The **ListNode** struct is a simple node of an ordinary **LinkedList**
        - **letter** (char)
        - **next** (pointer to ListNode): a pointer to the next node in the list
4. LinkedList:
    - The LinkedList just holds a pointer to a ListNode as its head
        - **head** (a pointer to a ListNode): a pointer to the head of the list
5. CircularLLNode:
    - The CircularLLNode is a node for CircularLinked lists (it is the same as ListNode but we separated it for convenience)
        - **letter** (char)
        - **next** (pointer to a CircularLLNode): a pointer to the next node in the list

6. CircularLLs:
    - The CircularLLs struct holds two heads for split Circular Linked Lists.
        - **head1** (pointer to a CircularLLNode): the head of the first split CircularLL
        - **head2** (pointer to a CircularLLNode): the head of the second split CircularLL

```
struct StationNode
{
    int id;
    float coordX;
    float coordY;
    StationNode* next;
};

struct Response
{
    char username[256];

    union Data
    {
        struct Error   {char code;} error;
        struct Success { float totalDistance; int stationCount; } success;
    } data;
};

struct ListNode
{
    char letter;
    ListNode* next;
};

struct LinkedList
{
    ListNode* head;
};

struct CircularLLNode
{
    char letter;
    CircularLLNode* next;
};


struct CircularLLs
```

```
{
    CircularLLNode* head1;
    CircularLLNode* head2;
};
```

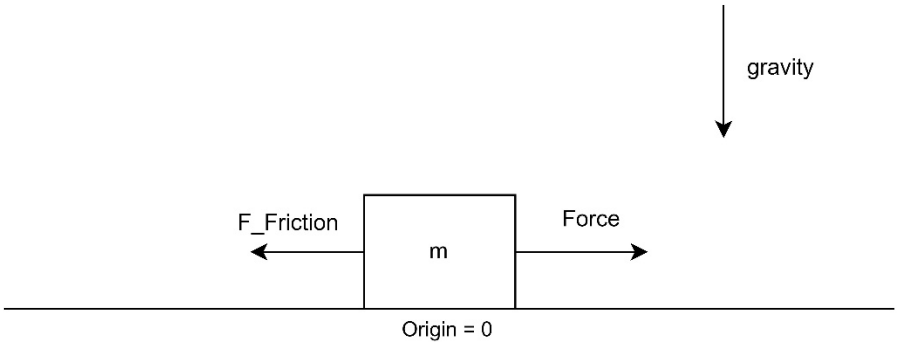
**Task 1 - CalculatePos(25 Pts)**

In this task, you will implement a function `void CalculatePos()` that reads input from **stdin** with **scanf** and prints the calculation:

You will take 5 **float** inputs in the same line that are separated with one blank space. These inputs are:

- force - N (float)
- mass - kg (float)
- gravity - m/s^2 (float)
- f_Coeff (float)
- time - s (float)

You will calculate an object's position where a Force is applied on a 1D line with friction:




By using these variables that you have read, you will calculate an object's position.

- F_Friction is calculated with **mass * gravity * f_Coeff**
- The acceleration of the object is calculated as :
  - **(Force - F_Friction) / mass**
- The position of the object will be:
  - **(1.0/2.0) * acceleration * time^2**

In the end, you will **print** the final position with 2-digit precision using **printf**

For example:

**Input:** 5.4 5.1 9.8 0.1 3.0

**Output:** The position is 0.35 (there is no newline at the end of the output)

**Input:** 20.4 10.1 9.8 0.7 2.0

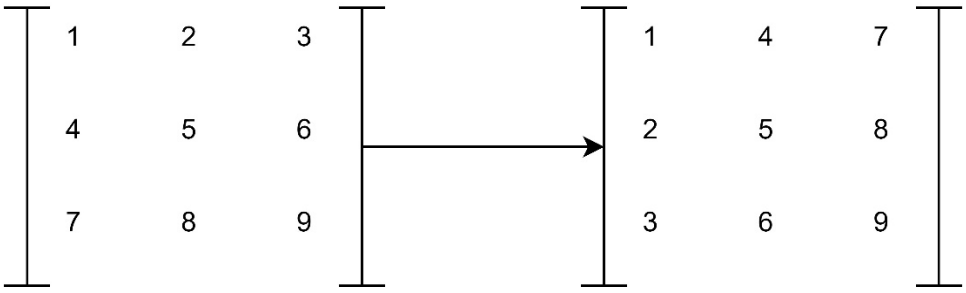
**Output:** The position is 0.00

**Notes:**

- Keep in mind that **F_Friction** cannot be bigger than **Force**. If that's the case, then the acceleration will be just 0.0 so the object does not move. Just like the second example
- The starting point of the object will be 0.0.

## Task 2 - TransposeSquareMatrix(25 pts)

In this task, you will implement a function `void TransposeSquareMatrix(int* mat, int n)` that reads a square nxn matrix and changes its elements to make the original matrix transpose.




In the picture above, you see how the transpose operation changes the matrix.

**Notes:**

- The matrices will always be **nxn square** matrices**.**
- **int* mat** argument will be a 1D array where matrix elements are stored in **row-major** order.
    - That means in the matrix on the above-left, the element (3rd row, 2nd column) is 8. It is accessed in a 1D array as:

        - mat[2 * 3 + 1] => 8 (the numbers are 2(3-1 th row) 3( n ) 1 (2-1 th column). You can access array elements like this.
- In this function, you don't need to print anything, just change the elements of the matrix accordingly.

**Example Mat:**

**1 2 9**

**3 2 1**

**4 4 4**

**Example TransposeMat**

**1 3 4**

**2 2 4**

**9 1 4**

## Task 3 - ProcessShipment(30 pts)

This is the same function that you have implemented in your lab preliminary work. But this time, the coordinates of each station are given in 2D.

You will implement a function `void ProcessShipment(char* username, float distanceLimit, StationNode* headNode, Response* resp)` that traverses the **Station List** given as the **headNode** and:

- Copy the username into the response struct given as **resp**.
- Iterate the stations and calculate the distance travelled between the first and the last node
- Calculate how many nodes you have counted as well.

If the distance you have calculated is bigger than the **distanceLimit:**

- Fill the **error** struct in **resp** with the code 'D'

If the distance is smaller or equal to the **distanceLimit:**

- Fill the **success** struct in **resp:**
    - stationCount will be how many nodes you have counted during the iteration
    - totalDistance will be the distance travelled that you have calculated.

**Note:**

The coordinates of the station nodes will be in 2D. So you have to compute the **Euclidean Distance** between the nodes.

Euclidean distance is calculated as the following:

- **sqrt( (coord1X - coord2X)^2 + (coord1Y - coord2Y)^2 )**

Example:

**username** = "Client1"

**distanceLimit** = 15

**Station list** => Station1 (0.8, 0.0) - Station 2 (1.2, 0.0) - Station3 (-3.5, 0.0) - Station4 (1.5, 0.0)

**Desired Response structure**

**resp**->username = "Client1"

**resp**->success.stationCount = 4

**resp**->success.totalDistance = 10.1

Example2:

**username** = "Client2"

**distanceLimit** = 9

**Station list** => Station1 (0.8, 0.0) - Station 2 (1.2, 0.0) - Station3 (-3.5, 0.0) - Station4 (1.5, 0.0)

**Desired Response structure**

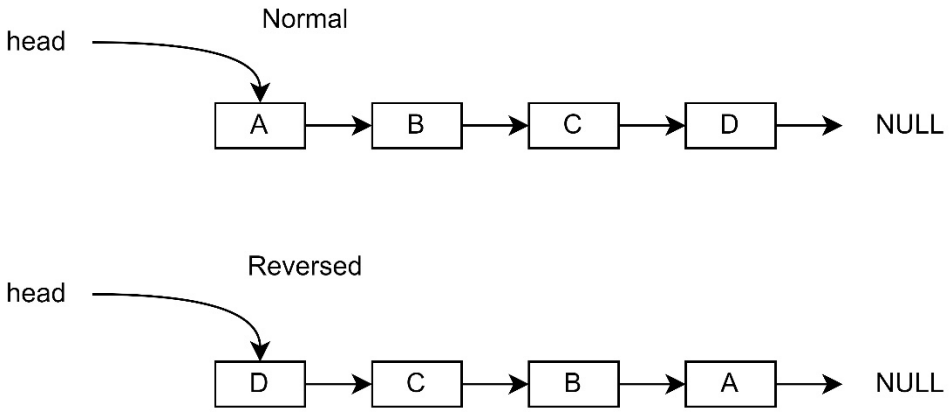
Note that this time, the distance travelled is greater that the distance limit, so we fill the error part of the union:

**resp**->data.error.code = 'D'

**Note:** Keep in mind that coordinates can have negative values. So they are between -inf, +inf

## Task 4 - ReverseLinkedList(10 pts)

You will implement the function `void ReverseLinkedList(LinkedList* list)` that reverses the list:




**Notes:**

- In this task, you will deal with the structs below:

```
struct ListNode
{
   char letter;
   ListNode* next;
};
```

```
struct LinkedList
{
    ListNode* head;
};
```

- **DO NOT** create new list nodes with malloc. You just have to modify the connections of list nodes and make the original list reversed.
- Do not forget to update the head node at the end so that the last element becomes the head in reversed version.
- The list will always have a size greater than or equal to **1**

## Task 5 - SplitCircularLLs(10 pts)

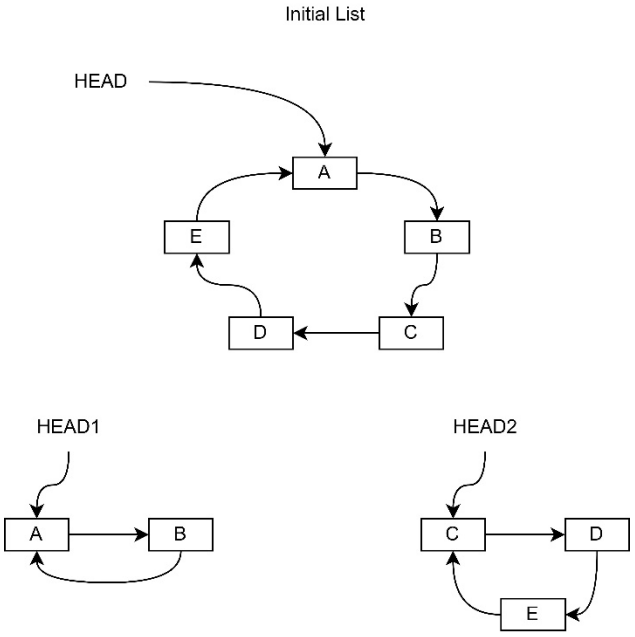
You will implement the function `CircularLLs* SplitCircularLLs(CircularLLNode* head)` that splits a circular linked list into two circular linked lists.

In this task, you will deal with the structs below:

```
struct CircularLLNode
{
    char letter;
    CircularLLNode* next;
};


struct CircularLLs
{
    CircularLLNode* head1;
    CircularLLNode* head2;
};
```

In the figure below, you can see how this split operation works.




**Notes:**

- In the circular linked list, node # will always be greater than 2.
- As is seen in the figure above, if we have an odd number of nodes, the first split circular linked list will have fewer nodes than the other.
- In your function, you will first allocate memory for **CircularLLs** and after you arrange the nodes, you will return a pointer to that struct
- You only need to **modify** the connections of each node such that we have two different circular linked lists at the end.

**Specifications:**

- Each task will be tested individually. You do NOT have to implement any other function to test a specific task. However, to **NOT get 0 points, your lab4.c implementation has to be bug-free which means it should be compiled without any error.**
- Please obey the print format for getting points in lab exam.
- You can use **strcpy**, **strcmp** functions from string.h
- **stdlib.h** and **string.h** libraries are included for you already.
- you can also use **math.h** functions as it is already included ofr you.
- you cannot use any other **library.**