1.IMPLEMENT STACK USING QUEUES

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

typedef struct StackItemBase {
    int value;
    struct StackItemBase* nextItemPtr;
} StackItem;

typedef struct {
    StackItem* head;
} MyStack;

MyStack* myStackCreate() {
    MyStack* newStack = (MyStack*)malloc(sizeof(MyStack));
    newStack->head = NULL;
    return newStack;
}

void myStackPush(MyStack* obj, int x) {
    StackItem* newHead = (StackItem*)malloc(sizeof(StackItem));
    newHead->value = x;
    newHead->nextItemPtr = obj->head;
    obj->head = newHead;
}

int myStackPop(MyStack* obj) {
    int result  = obj->head->value;
    StackItem* newHead = obj->head->nextItemPtr;
    free(obj->head);
    obj->head = newHead;
    return result;
}

int myStackTop(MyStack* obj) {
    return obj->head->value;
}

bool myStackEmpty(MyStack* obj) {
    return obj->head == NULL;
}

void myStackFree(MyStack* obj) {
    while (obj->head != NULL) {
        StackItem* newHead = obj->head->nextItemPtr;
        free(obj->head);
        obj->head = newHead;
    }
    free(obj);
}
```

Problem List   Run   Submit   0   Premium

Description | Editorial | Solutions | Submissions

225. Implement Stack using Queues

Solved

Easy   Topics   Companies

Implement a last-in-first-out (LIFO) stack using only two queues. The implemented stack should support all the functions of a normal stack (`push`, `top`, `pop`, and `empty`).

Implement the `MyStack` class:

- `void push(int x)` Pushes element x to the top of the stack.
- `int pop()` Removes the element on the top of the stack and returns it.
- `int top()` Returns the element on the top of the stack.
- `boolean empty()` Returns `true` if the stack is empty, `false` otherwise.

Notes:

- You must use **only** standard operations of a queue, which means that only `push to back`, `peek/pop from front`, `size` and `is empty` operations are valid.
- Depending on your language, the queue may not be supported natively. You may simulate a queue using a list or deque (double-ended queue) as long as you use only a queue's standard operations.

Example 1:

5.9K   46

Code

C   Auto

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

typedef struct StackItemBase {
    int value;
    struct StackItemBase* nextItemPtr;
} StackItem;

typedef struct {
    StackItem* head;
} MyStack;

MyStack* myStackCreate() {
    MyStack* newStack = (MyStack*)malloc(sizeof(MyStack));
    newStack->head = NULL;
```

Saved to local   Ln 1, Col 1

Testcase | Test Result

Accepted   Runtime: 4 ms

Case 1

Input

```
["MyStack","push","push","top","pop","empty"]
```

2.IMPLEMENT QUEUE USING STACKS

```
#include<stdio.h>
typedef struct {
  int s1[100];
  int s2[100];
  int cnt1;
  int cnt2;
} MyQueue;

MyQueue* myQueueCreate()
{
  MyQueue* que = malloc(sizeof(MyQueue));
  que->cnt1 = 0;
  que->cnt2 = 0;
  return que;
}

void myQueuePush(MyQueue* obj, int x)
{
     obj->s1[obj->cnt1++] = x;
}

int myQueuePop(MyQueue* obj)
{
  if (obj->cnt1 == 0 && obj->cnt2 == 0) return -1;
  if (obj->cnt2 == 0)
  {
    while (obj->cnt1 > 0)
    {
      obj->s2[obj->cnt2++] = obj->s1[--obj->cnt1];
    }
  }
  int a = obj->s2[--obj->cnt2];
  return a;
}

int myQueuePeek(MyQueue* obj)
{
  if (obj->cnt2 == 0) return obj->s1[0];
  return obj->s2[obj->cnt2 - 1];
}

bool myQueueEmpty(MyQueue* obj)
{
  if (obj->cnt1 == 0 && obj->cnt2 == 0) return true;
  return false;
}

void myQueueFree(MyQueue* obj)
{
    free(obj);
}
```

Problem List Run Submit Premium

Description | Editorial | Solutions | Submissions

232. Implement Queue using Stacks

Easy · Topics · Companies

Implement a first in first out (FIFO) queue using only two stacks. The implemented queue should support all the functions of a normal queue (`push`, `peek`, `pop`, and `empty`).

Implement the `MyQueue` class:

- `void push(int x)` Pushes element x to the back of the queue.
- `int pop()` Removes the element from the front of the queue and returns it.
- `int peek()` Returns the element at the front of the queue.
- `boolean empty()` Returns `true` if the queue is empty, `false` otherwise.

Notes:

- You must use **only** standard operations of a stack, which means only `push to top`, `peek/pop from top`, `size`, and `is empty` operations are valid.
- Depending on your language, the stack may not be supported natively. You may simulate a stack using a list or deque (double-ended queue) as long as you use only a stack's standard operations.

Example 1:

7.4K · 89

Code

C · Auto

```c
#include<stdio.h>
typedef struct {
  int s1[100];
  int s2[100];
  int cnt1;
  int cnt2;
} MyQueue;

MyQueue* myQueueCreate()
{
  MyQueue* que = malloc(sizeof(MyQueue));
  que->cnt1 = 0;
  que->cnt2 = 0;
  return que;
}
```

Saved to local · Ln 1, Col 1

Testcase | Test Result

Accepted Runtime: 0 ms

Case 1

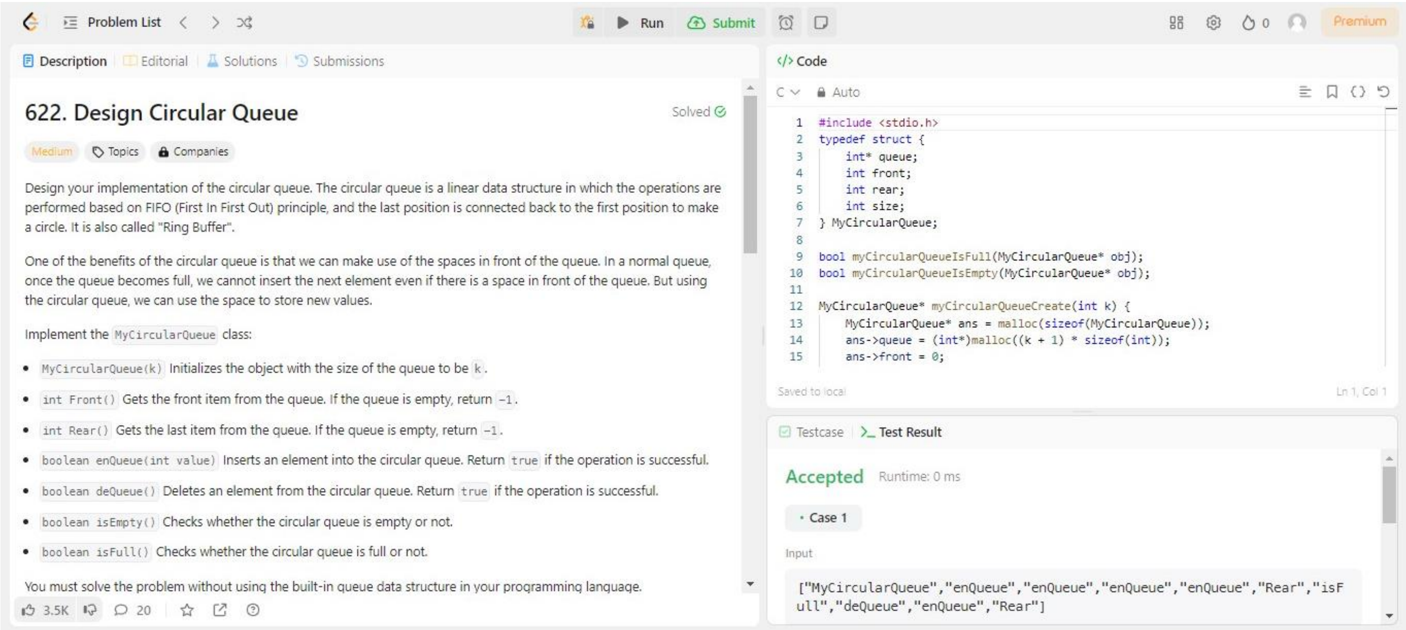
Input

["MyQueue","push","push","peek","pop","empty"]

3.DESIGN CIRCULAR QUEUE

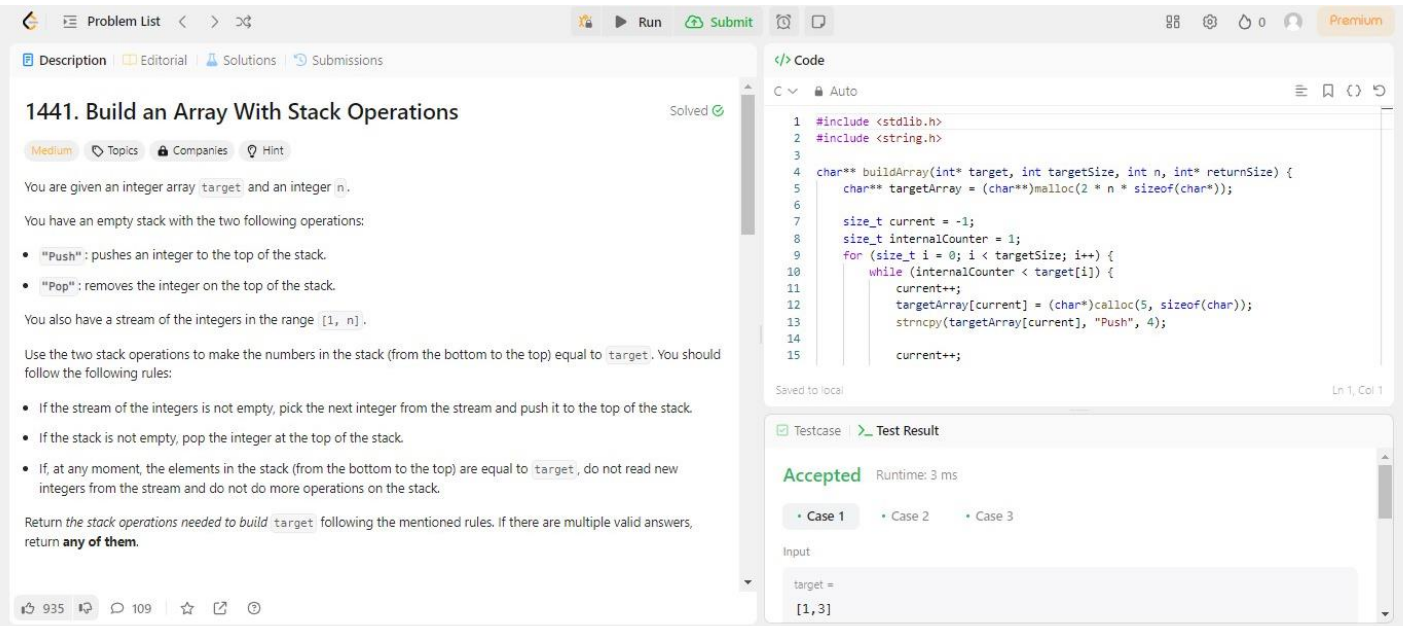
```
#INCLUDE<STDIO.H>
typedef struct
{
    int* queue;
    int front;
    int rear;
    int size;
} MyCircularQueue;

bool myCircularQueueIsFull(MyCircularQueue* obj);
bool myCircularQueueIsEmpty(MyCircularQueue* obj);

MyCircularQueue* myCircularQueueCreate(int k)
{
    MyCircularQueue* ans = malloc(sizeof(MyCircularQueue));
    ans->queue = (int*)malloc((k+1) * sizeof(int));
    ans->front = 0;
    ans->rear = 0;
    ans->size = k+1;
    return ans;
}

bool myCircularQueueEnQueue(MyCircularQueue* obj, int value)
{

    if(myCircularQueueIsFull(obj))
        return false;
    else{
        obj->queue[obj->rear] = value;
        obj->rear = (obj->rear+1) % (obj->size);
        return true;
    }
}

bool myCircularQueueDeQueue(MyCircularQueue* obj)
{

    if(myCircularQueueIsEmpty(obj))
        return false;
    else{
        obj->front = (obj->front + 1)% obj->size;
        return true;
    }
}

int myCircularQueueFront(MyCircularQueue* obj)
{
    if(myCircularQueueIsEmpty(obj))
        return -1;
    else
        return obj->queue[obj->front];
}

int myCircularQueueRear(MyCircularQueue* obj)
{
    if(myCircularQueueIsEmpty(obj))
        return -1;
    else
        return obj->queue[(obj->rear-1+obj->size) % obj->size];
}

bool myCircularQueueIsEmpty(MyCircularQueue* obj)
{

    if((obj->front) == (obj->rear))
        return true;
    else
        return false;
}

bool myCircularQueueIsFull(MyCircularQueue* obj)
```

```
{
    return ( (obj->rear + 1) % obj->size == obj->front);
}

void myCircularQueueFree(MyCircularQueue* obj)
{
    free(obj->queue);
    free(obj);
}
```



4.BUILD ARRAY USING STACK OPERATIONS

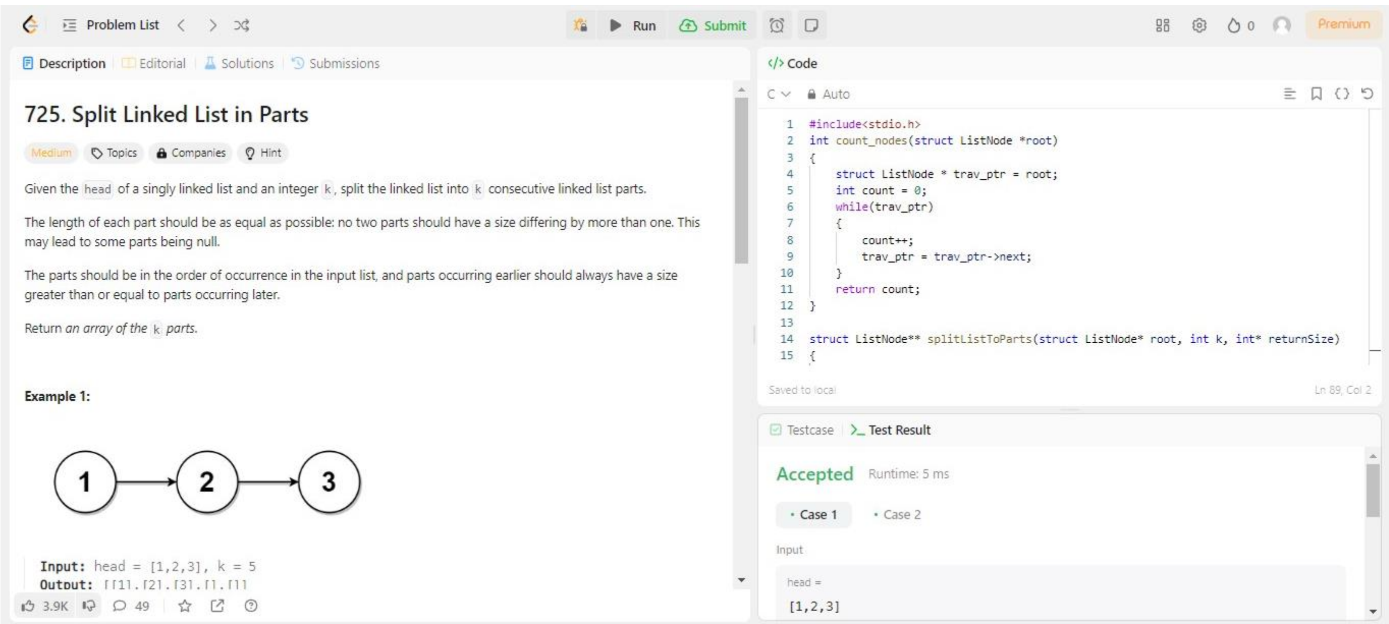
```
#include <stdlib.h>

#include <string.h>


char** buildArray(int* target, int targetSize, int n, int* returnSize) {
    char** targetArray = (char**)malloc(2 * n * sizeof(char*));

    size_t current = -1;
    size_t internalCounter = 1;
    for (size_t i = 0; i < targetSize; i++) {
        while (internalCounter < target[i]) {
            current++;
            targetArray[current] = (char*)calloc(5, sizeof(char));
            strncpy(targetArray[current], "Push", 4);

            current++;
            targetArray[current] = (char*)calloc(5, sizeof(char));
            strncpy(targetArray[current], "Pop", 4);

            internalCounter++;
        }

        current++;
        targetArray[current] = (char*)calloc(5, sizeof(char));
        strncpy(targetArray[current], "Push", 4);

        if (target[targetSize - 1] == internalCounter) {
            break;
        }

        internalCounter++;
    }

    *returnSize = current + 1;
    return targetArray;
}
```



5.SPLIT LINKED LIST IN PARTS

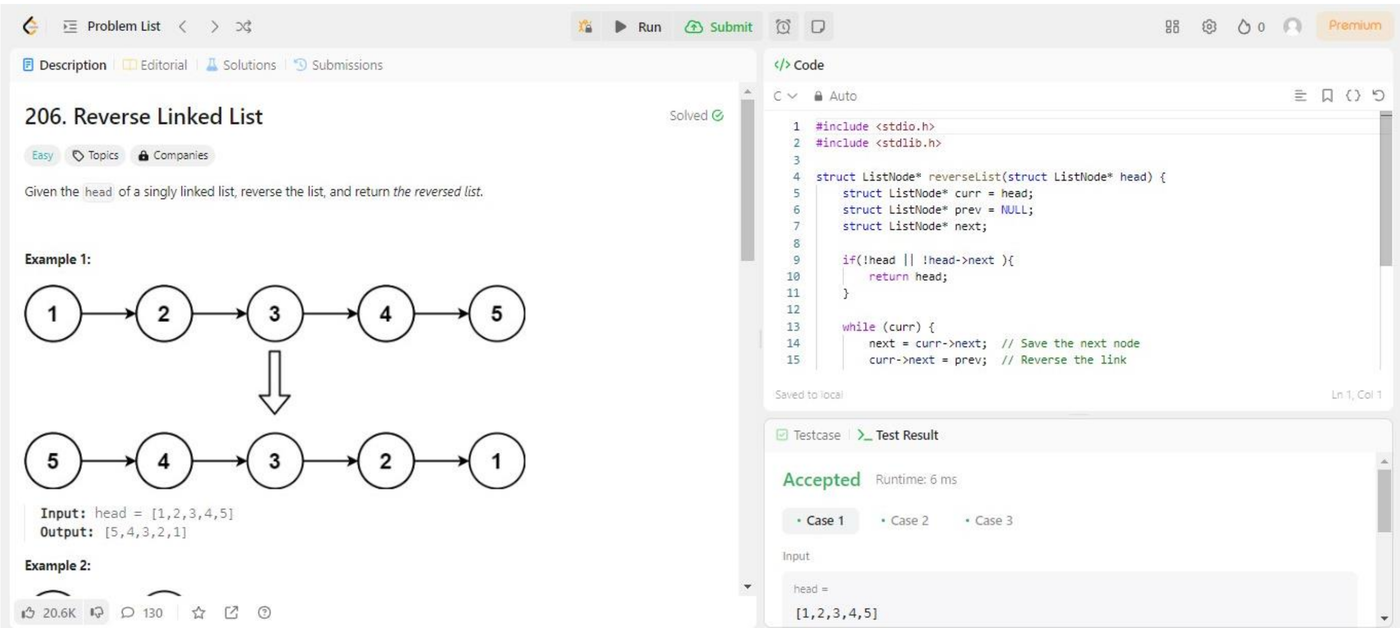
```
#include<stdio.h>
int count_nodes(struct ListNode *root)
{
    struct ListNode * trav_ptr = root;
    int count = 0;
    while(trav_ptr)
    {
        count++;
        trav_ptr = trav_ptr->next;
    }
    return count;
}

struct ListNode** splitListToParts(struct ListNode* root, int k, int* returnSize)
{
    int no_of_nodes = count_nodes(root);

    int no_of_elements_in_each_half;
    int rem;

    if( k <= no_of_nodes)
    {
      no_of_elements_in_each_half  = (no_of_nodes/k);
      rem = (no_of_nodes%k);
    }
    else
    {
        no_of_elements_in_each_half = 1;
        rem = 0;
    }

    struct ListNode ** list_ptr = (struct ListNode **)malloc(sizeof(struct ListNode *)*k);

    int i=0;
    for(i=0; i<k; i++)
    {
        memset((list_ptr+i), NULL, sizeof(struct ListNode *));
    }

    int list_ptr_index = 0;

    struct ListNode * trav_ptr = root;
    struct ListNode * next_ptr = root;
    int count = 1;

    while(trav_ptr)
    {
       if(count == no_of_elements_in_each_half)
       {
           if(rem != 0)
           {
               rem--;
               count = 1;
               trav_ptr = trav_ptr->next;
               next_ptr = trav_ptr->next;
               trav_ptr->next = NULL;
               *(list_ptr + list_ptr_index) = root;
               list_ptr_index++;

               root = next_ptr;
               trav_ptr = next_ptr;
           }
           else
           {
```

```
            count = 1;
            next_ptr = trav_ptr->next;
            trav_ptr->next = NULL;

            *(list_ptr + list_ptr_index) = root;
            list_ptr_index++;

            root = next_ptr;
            trav_ptr = next_ptr;
        }
    }
    else
    {
        count++;
        trav_ptr = trav_ptr->next;
    }
}


     *returnSize = k;

    return list_ptr;


}
```



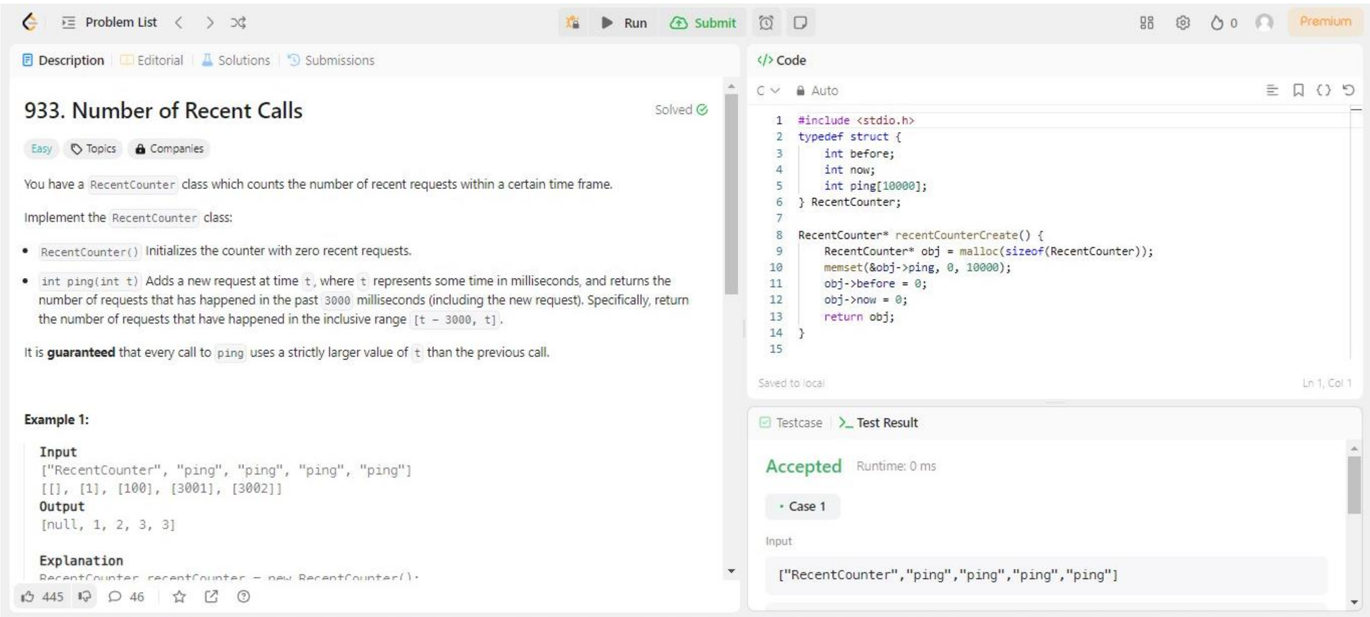
6.REVERSE LINKED LIST

```
#include<stdio.h>
struct ListNode* reverseList(struct ListNode* head)
{

    struct ListNode* temp=NULL;
    struct ListNode* previous=NULL;

    while(head!=NULL)
    {
        temp=head->next;
        head->next=previous;
        previous=head;
        head=temp;
    }
    return previous;
}
```



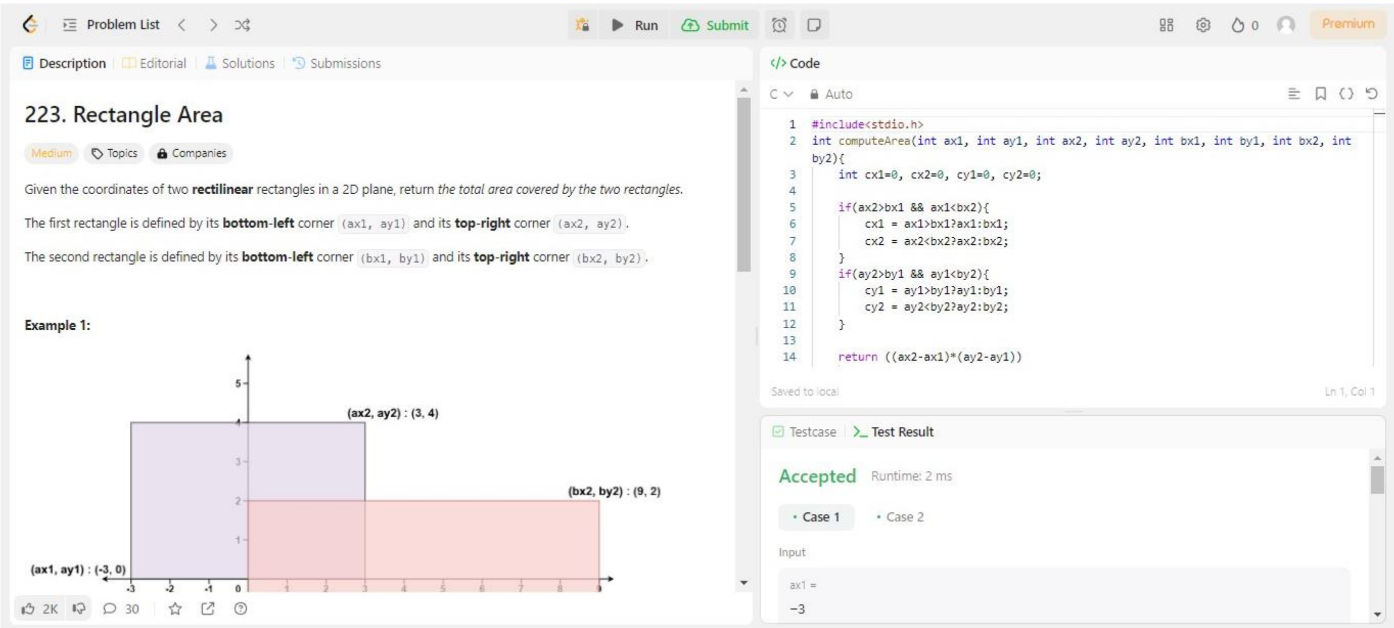
7.NUMBER OF RECENT CALLS

```c
#include<stdio.h>
typedef struct
{
    int before;
    int now;
    int ping[10000];
} RecentCounter;

RecentCounter *recentCounterCreate()
{
    RecentCounter *obj = malloc(sizeof(RecentCounter));
    memset(&obj->ping, 0, 10000);
    obj->before = 0;
    obj->now = 0;
    return obj;
}

int recentCounterPing(RecentCounter *obj, int t)
{
    obj->ping[obj->now++] = t;
    while (obj->ping[obj->before] < t - 3000) obj->before++;
    return obj->now - obj->before;
}

void recentCounterFree(RecentCounter *obj)
{
    free(obj);
}
```



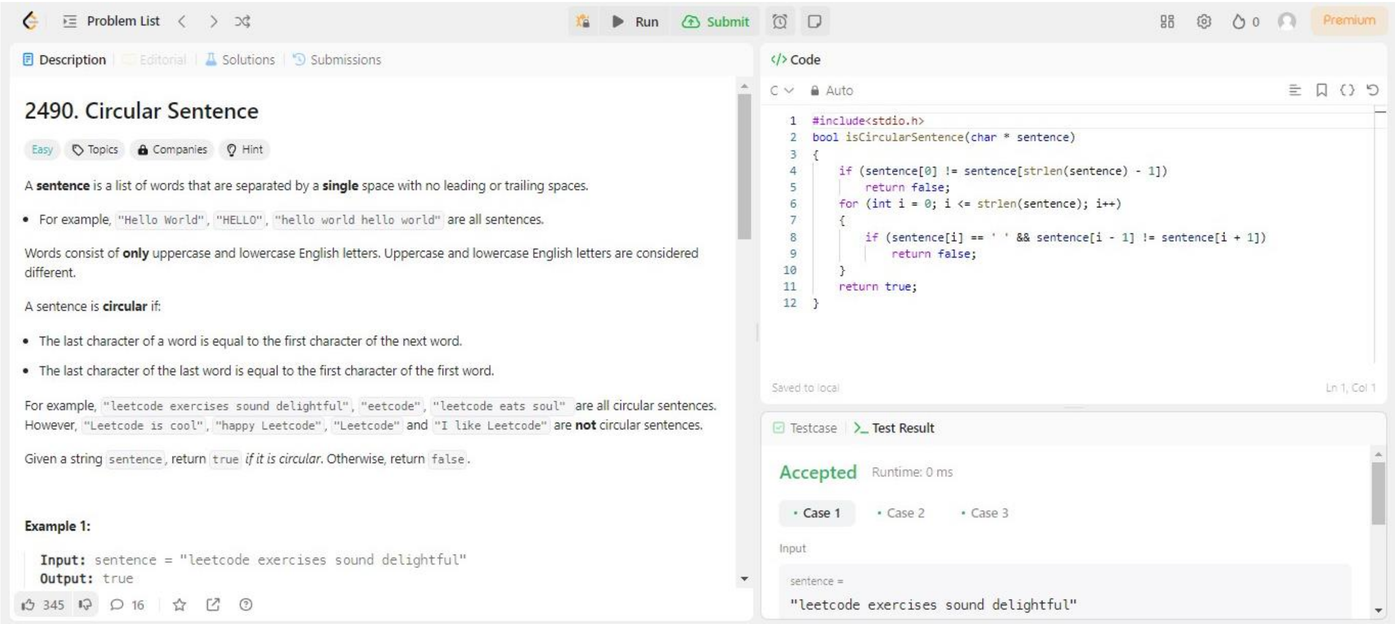
8.RECTANGLE AREA

```c
#include<stdio.h>
int computeArea(int ax1, int ay1, int ax2, int ay2, int bx1, int by1, int bx2, int by2){
    int cx1=0, cx2=0, cy1=0, cy2=0;

    if(ax2>bx1 && ax1<bx2){
        cx1 = ax1>bx1?ax1:bx1;
        cx2 = ax2<bx2?ax2:bx2;
    }
    if(ay2>by1 && ay1<by2){
        cy1 = ay1>by1?ay1:by1;
        cy2 = ay2<by2?ay2:by2;
    }

    return ((ax2-ax1)*(ay2-ay1))
        + ((bx2-bx1)*(by2-by1))
        - ((cx2-cx1)*(cy2-cy1));
}
```



9.CIRCULAR SENTENCE

```
#include<stdio.h>
bool isCircularSentence(char * sentence)
{
    if (sentence[0] != sentence[strlen(sentence) - 1])
        return false;
    for (int i = 0; i <= strlen(sentence); i++)
    {
        if (sentence[i] == ' ' && sentence[i - 1] != sentence[i + 1])
            return false;
    }
    return true;
}
```



10.BINARY TREE PATHS

```
#include<stdio.h>

int resIdx;
void findpath (struct TreeNode* root, int *ls,int ls_idx,char **res);
char ** binaryTreePaths(struct TreeNode* root, int* returnSize)
{

    if (root == NULL)
    {
        *returnSize = 0;
        return NULL;
    }
    resIdx = 0;
    int i;
    char ** res = (char **)malloc(sizeof(char *) * 40);
    int ls[150];
    findpath(root,&ls[0],0,res);
    *returnSize = resIdx;
    return &res[0];

}


void findpath (struct TreeNode* root, int *ls,int ls_idx,char **res)
{
    char temp[100];
    int l=0,i=0;
    if (root->left == NULL && root->right == NULL)
    {
        ls[ls_idx] = root->val;
        ls_idx+=1;
        res[resIdx] = (char *)malloc(sizeof(char) * 100);
        while (i < ls_idx)
        {
            if (i==0)
            {
                l = l + sprintf(&temp[l], "%d", ls[i]);

            }
            else
            {
                l = l + sprintf(&temp[l], "->%d", ls[i]);
            }
            i++;
        }
        strcpy(res[resIdx],temp);
        resIdx++;
        return;
    }
    ls[ls_idx] = root->val;
    if (root->left != NULL)
    {
        findpath(root->left,ls,ls_idx+1,res);
    }

    if (root->right != NULL)
    {
        findpath(root->right,ls,ls_idx+1,res);
    }
    return;

}
```

Problem List   Run   Submit   Premium

Description | Editorial | Solutions | Submissions

257. Binary Tree Paths

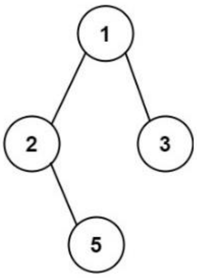
Solved

Easy   Topics   Companies

Given the `root` of a binary tree, return *all root-to-leaf paths in* ***any order***.

A **leaf** is a node with no children.

Example 1:



6.4K   28

Code

C   Auto

```c
#include<stdio.h>
int resIdx;
void findpath (struct TreeNode* root, int *ls,int ls_idx,char **res);
char ** binaryTreePaths(struct TreeNode* root, int* returnSize)
{

    if (root == NULL)
    {
        *returnSize = 0;
        return NULL;
    }
    resIdx = 0;
    int i;
    char ** res = (char **)malloc(sizeof(char *) * 40);
    int ls[150];
```

Saved to local   Ln 1, Col 1

Testcase   >_ Test Result

Accepted   Runtime: 2 ms

Case 1   Case 2

Input

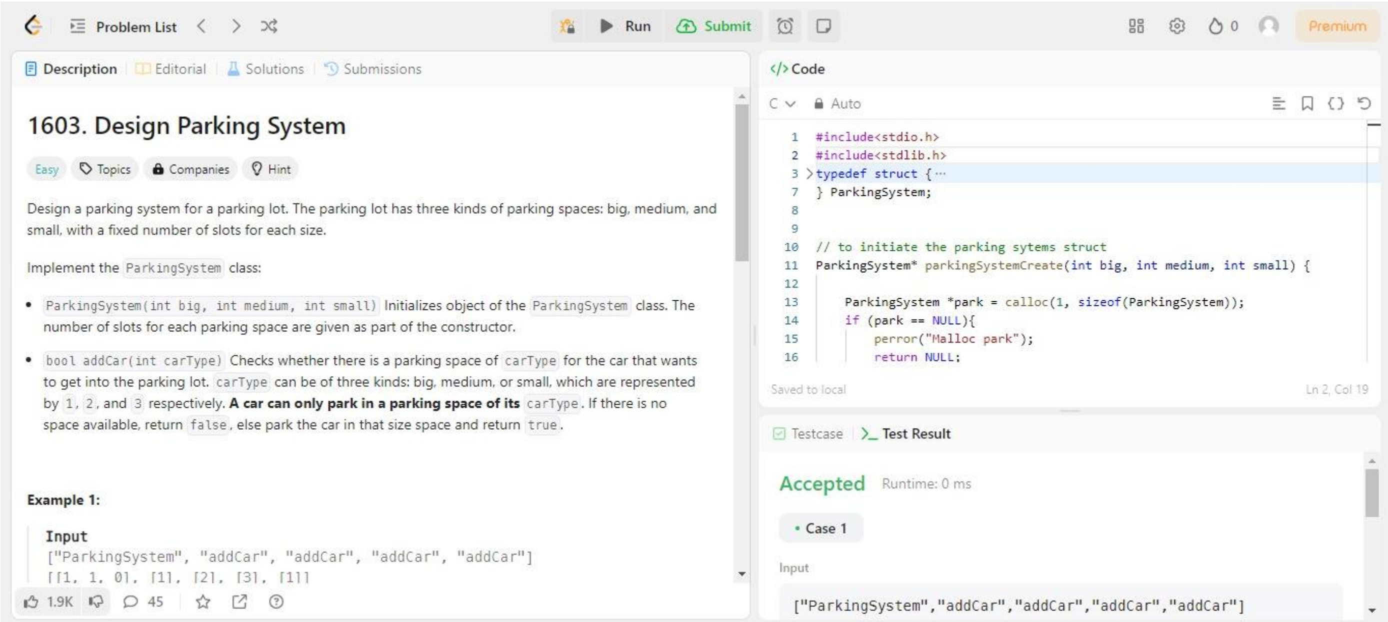
root =

```
[1,2,3,null,5]
```

PROBLEM: **DESIGN A PARKING SYSTEM**

```
#include<stdio.h>
#include<stdlib.h>
typedef struct {
    int big;
    int medium;
    int small;
} ParkingSystem;


// to initiate the parking sytems struct
ParkingSystem* parkingSystemCreate(int big, int medium, int small) {

    ParkingSystem *park = calloc(1, sizeof(ParkingSystem));
    if (park == NULL){
        perror("Malloc park");
        return NULL;
    }

    // assigning the slot size for each parking lot type
    park->big = big;
    park->medium = medium;
    park->small = small;

    return park;
}

// Add the car to the parking lot. Return true if its able, false otherwise.
bool parkingSystemAddCar(ParkingSystem* obj, int carType) {

    // If the carType is big, reduce the big slot
    if (carType == 1 && obj->big != 0){
        obj->big -= 1;
        return true;
    }

    // If the carType is medium, reduce the medium slot
    if (carType == 2 && obj->medium != 0){
        obj->medium -= 1;
        return true;
    }

    // If the carType is small, reduce the small slot
    if (carType == 3 && obj->small != 0){
        obj->small -= 1;
        return true;
    }



    return false;
}


// Free the malloc
void parkingSystemFree(ParkingSystem* obj) {
```

```c
    free(obj);
}

/**
 * Your ParkingSystem struct will be instantiated and called as such:
 * ParkingSystem* obj = parkingSystemCreate(big, medium, small);
 * bool param_1 = parkingSystemAddCar(obj, carType);

 * parkingSystemFree(obj);
*/
```



A "real-time problem on queues" could mean a few things, but in a general sense, it could refer to a problem that is designed to mimic a real-world scenario where a queue data structure might be used. Queues are commonly used in real-world applications for tasks like job scheduling, handling tasks in a first-come-first-served (FIFO) manner, or managing resources like printers in an office.

Here is a simple example of a "real-time problem on queues"

Problem: **Design a Parking System**

Design a parking system for a parking lot that has three kinds of parking spaces: **big**, **medium**, and **small**, with respective capacities of **5**, **10**, and **20** parking spaces. Implement the ParkingSystem class:

- ParkingSystem(int big, int medium, int small) Initializes object of the ParkingSystem class. The number of each type of space is given as **big**, **medium**, and **small**, respectively.
- **bool addCar(int carType)** Checks whether there is a parking space of carType for the car that wants to get into the parking lot. carType can be of three kinds: **big**, **medium**, or **small**. The parking lot has at least one empty parking space for each car type.

You need to return true if there is a parking space for the car, otherwise, return false.

Input:

["ParkingSystem", "addCar", "addCar", "addCar", "addCar"] [[1, 1, 0], [1], [2], [3], [1]]

Output:

[null, true, true, false, false]

Explanation:

ParkingSystem parkingSystem = new ParkingSystem(1, 1, 0); parkingSystem.addCar(1); // return true because there is 1 available slot for a big car parkingSystem.addCar(2); // return true because there is 1 available slot for a medium car parkingSystem.addCar(3); // return false because there is no available slot for a small car parkingSystem.addCar(1); // return false because there is no available slot for a big car. It is already occupied.

Note:

- The number of spaces is in the range [0, 1000]
- carType is 1, 2, or 3
- Each carType may be either a big car, medium car, or a small car

This problem simulates a real-world scenario where a queue data structure might be used to manage parking space availability. It requires implementing a queue to track the available parking spaces for each car type and to check if there is an available parking space for a given car type when a car arrives at the parking lot.