ADVANCE CODING

Assignment 2

K.Srikar

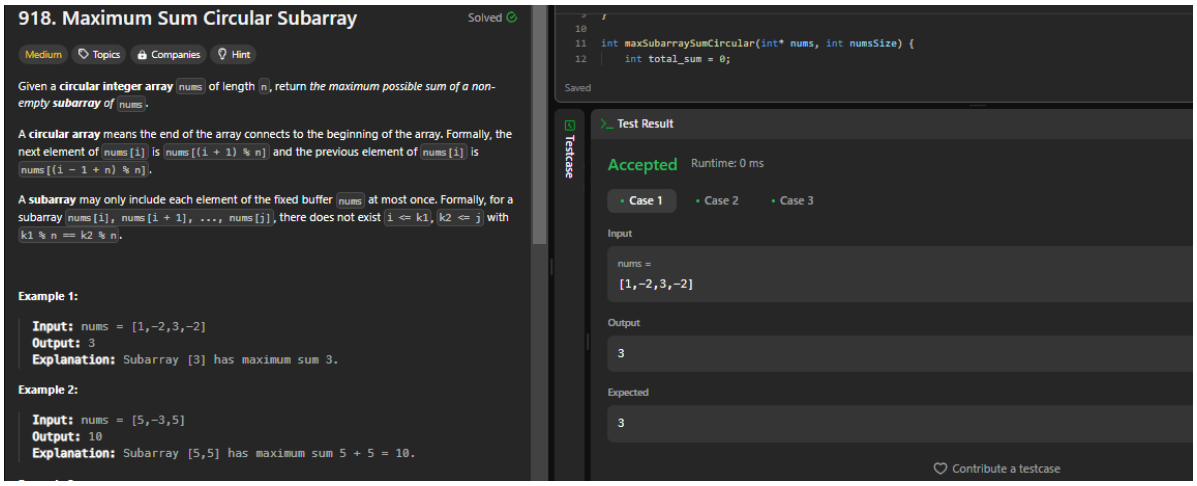
VU21CSEN0300058

Maximum Sum Circular Subarray

Code:

```c
#include <limits.h>
int max(int a, int b) {
    return (a > b) ? a : b;
}
int min(int a, int b) {
    return (a < b) ? a : b;
}
int maxSubarraySumCircular(int* nums, int numsSize) {
    int total_sum = 0;
    int max_sum = INT_MIN, curr_max = 0;
    int min_sum = INT_MAX, curr_min = 0;

    for (int i = 0; i < numsSize; i++) {
        total_sum += nums[i];
        curr_max = max(curr_max + nums[i], nums[i]);
        max_sum = max(max_sum, curr_max);
        curr_min = min(curr_min + nums[i], nums[i]);
        min_sum = min(min_sum, curr_min);
    }
    if (max_sum < 0) {
        return max_sum;
    }
    return max(max_sum, total_sum - min_sum);
}
```

Output:



Stamping the Sequence:

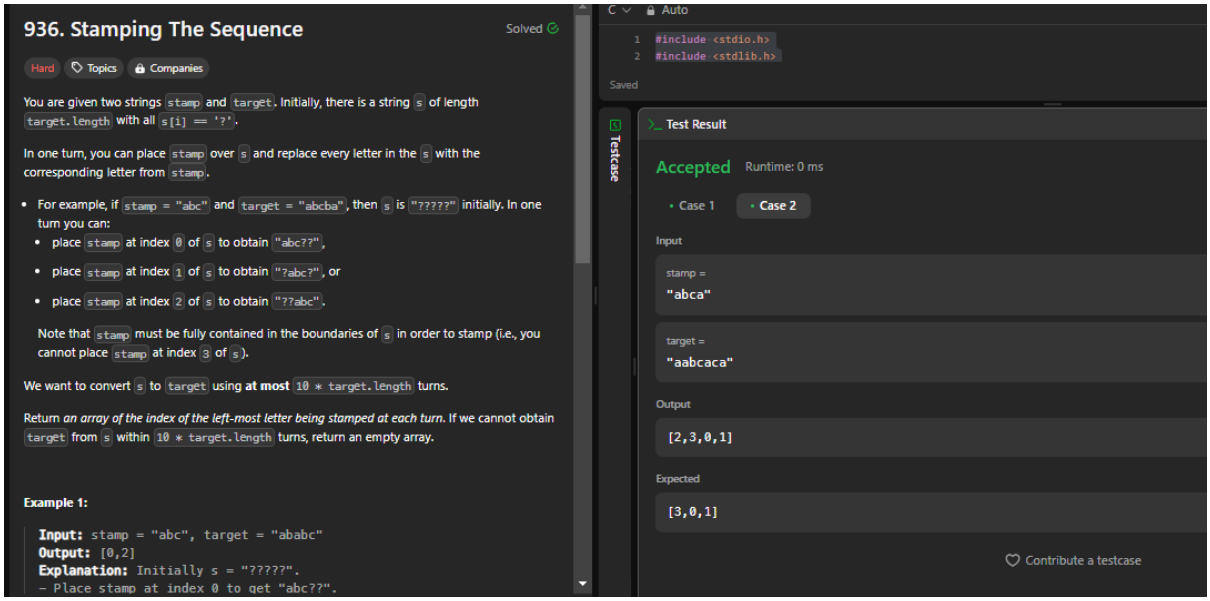
Code:

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

typedef struct {
    int* made;
    int* todo;
    int madeSize;
    int todoSize;
} Node;

void push(int* stack, int* top, int value) {
    stack[(*top)++] = value;
}

int* movesToStamp(char* stamp, char* target, int* returnSize) {
    int M = strlen(stamp), N = strlen(target);
    int* queue = (int*)malloc(N * sizeof(int));
    int queueFront = 0, queueRear = 0;
    bool* done = (bool*)calloc(N, sizeof(bool));
    int* ans = (int*)malloc(N * sizeof(int));
    int ansIndex = 0;
    Node* A = (Node*)malloc(N * sizeof(Node));

    for (int i = 0; i < N; ++i) {
        A[i].made = (int*)malloc(M * sizeof(int));
        A[i].todo = (int*)malloc(M * sizeof(int));
        A[i].madeSize = 0;
        A[i].todoSize = 0;
    }

    for (int i = 0; i <= N - M; ++i) {
        for (int j = 0; j < M; ++j) {
            if (target[i + j] == stamp[j]) {
                A[i].made[A[i].madeSize++] = i + j;
            } else {
                A[i].todo[A[i].todoSize++] = i + j;
            }
        }
        if (A[i].todoSize == 0) {
            push(ans, &ansIndex, i);
            for (int j = i; j < i + M; ++j) {
                if (!done[j]) {
                    queue[queueRear++] = j;
                    done[j] = true;
                }
            }
        }
    }

    while (queueFront < queueRear) {
```

```
        int i = queue[queueFront++];
        for (int j = fmax(0, i - M + 1); j <= fmin(N - M, i); ++j) {
            bool affected = false;
            for (int k = 0; k < A[j].todoSize; ++k) {
                if (A[j].todo[k] == i) {
                    affected = true;
                    break;
                }
            }
            if (affected) {
                for (int k = 0; k < A[j].todoSize; ++k) {
                    if (A[j].todo[k] == i) {
                        A[j].todo[k] = A[j].todo[A[j].todoSize - 1];
                        A[j].todoSize--;
                        break;
                    }
                }
                if (A[j].todoSize == 0) {
                    push(ans, &ansIndex, j);
                    for (int k = 0; k < A[j].madeSize; ++k) {
                        int m = A[j].made[k];
                        if (!done[m]) {
                            queue[queueRear++] = m;
                            done[m] = true;
                        }
                    }
                }
            }
        }
    }

    for (int i = 0; i < N; ++i) {
        if (!done[i]) {
            *returnSize = 0;
            free(queue);
            free(done);
            free(ans);
            for (int i = 0; i < N; i++) {
                free(A[i].made);
                free(A[i].todo);
            }
            free(A);
            return NULL;
        }
    }

    int* result = (int*)malloc(ansIndex * sizeof(int));
    for (int i = 0; i < ansIndex; i++) {
        result[i] = ans[ansIndex - 1 - i];
    }

    *returnSize = ansIndex;
    free(queue);
    free(done);
```

```
    free(ans);
    for (int i = 0; i < N; i++) {
        free(A[i].made);
        free(A[i].todo);
    }
    free(A);
    return result;
}
```

Output:



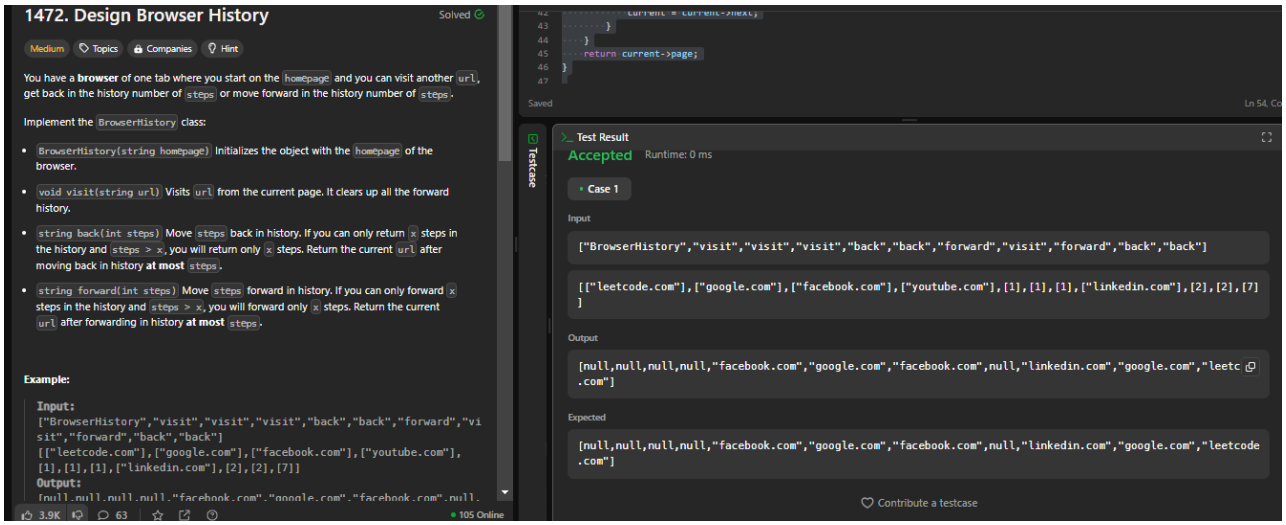
Design Browser History:

Code:

```
typedef struct BrowserHistory {
    char *page;
    struct BrowserHistory *next;
    struct BrowserHistory *prev;
} BrowserHistory;
BrowserHistory *current
BrowserHistory* browserHistoryCreate(char * homepage) {
    current = (BrowserHistory*) malloc(sizeof(BrowserHistory));
    current->page = homepage;
    current->next = NULL;
    current->prev = NULL;
    return current;
}

void browserHistoryVisit(BrowserHistory* obj, char * url) {
    //browserHistoryFree(current->next);
    current->next = malloc(sizeof(BrowserHistory));
    current->next->page = url;
    current->next->prev = current;
    current->next->next = NULL;
```

```
    current = current->next;
}
char * browserHistoryBack(BrowserHistory* obj, int steps) {
    for (int i = 0; i < steps; i++) {
        if (current->prev == NULL) {
            current = obj;
        } else {
            current = current->prev;
        }
    }
    return current->page;
}
char * browserHistoryForward(BrowserHistory* obj, int steps) {
    for (int i = 0; i < steps; i++) {
        if (current->next == NULL) {
            break;
        } else {
            current = current->next;
        }
    }
    return current->page;
}
void browserHistoryFree(BrowserHistory* obj) {
    if (obj == NULL) {
        return;
    }
    browserHistoryFree(obj->next);
    free(obj);
}
```

Output:



LRU Cache:

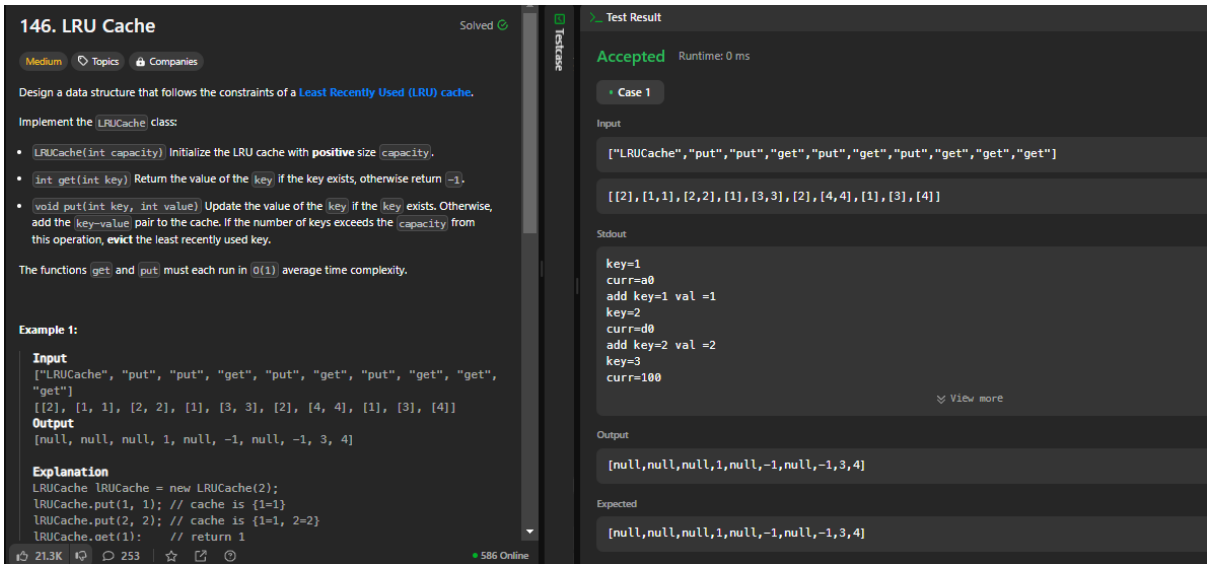
Code:

```
struct node {
    int key;
    int val;
```

```
    struct node *next;
    struct node *prev;
};
typedef struct {
    int cap;
    int count;
} LRUCache;
struct node *head;
struct node *tail;
struct node* rem[10001];
LRUCache* lRUCacheCreate(int capacity) {

    LRUCache *cache = malloc(sizeof(LRUCache));
    cache->cap = capacity;
    cache->count = 0;
    head = malloc(sizeof(struct node));
    tail = malloc(sizeof(struct node));

    head->next = tail;
    tail->prev = head;

    for (int i = 0; i < 10001; i++)
        rem[i] = NULL;

    return cache;
}

void del(struct node *curr) {

    curr->prev->next = curr->next;
    curr->next->prev = curr->prev;
}

void add(struct node *curr) {

    curr->next = head->next;
    curr->prev = head;

    head->next->prev = curr;
    head->next = curr;


}

int lRUCacheGet(LRUCache* obj, int key) {
        if(obj->count == 0)
            return -1;

        if(rem[key] == NULL)
            return -1;

        int val;
```

```
        struct node *curr = rem[key];
        val = curr->val;
        del(curr);
        add(curr);
        return val;
}

void lRUCachePut(LRUCache* obj, int key, int value) {

        printf("key=%d\n",key);
        if(rem[key] != NULL) {

        struct node *curr = rem[key];
        curr->val = value;
        del(curr);
        add(curr);
        }

        else {

            if(obj->count == obj->cap) {
                obj->count--;
                rem[tail->prev->key] = NULL;
                del(tail->prev);
            }

            obj->count++;
            struct node *curr = malloc(sizeof(struct node));
            curr->val = value;
            curr->key = key;
            printf("curr=%x\n",curr);
            printf("add key=%d val =%d \n",key,value);
            rem[key] = curr;
            add(curr);
        }

}

void lRUCacheFree(LRUCache* obj) {



    for(int i = 0; i <= 10000; i++)
        if(rem[i])
            free(rem[i]);
    free(head);
    free(tail);
    free(obj);
}
```

Output:



146. LRU Cache

Solved

Medium Topics Companies

Design a data structure that follows the constraints of a Least Recently Used (LRU) cache.

Implement the `LRUCache` class:

- `LRUCache(int capacity)` Initialize the LRU cache with **positive** size `capacity`.
- `int get(int key)` Return the value of the `key` if the key exists, otherwise return `-1`.
- `void put(int key, int value)` Update the value of the `key` if the `key` exists. Otherwise, add the `key-value` pair to the cache. If the number of keys exceeds the `capacity` from this operation, **evict** the least recently used key.

The functions `get` and `put` must each run in `O(1)` average time complexity.

Example 1:

Input
["LRUCache", "put", "put", "get", "put", "get", "put", "get", "get", "get"]
[[2], [1, 1], [2, 2], [1], [3, 3], [2], [4, 4], [1], [3], [4]]
Output
[null, null, null, 1, null, -1, null, -1, 3, 4]

Explanation
```
LRUCache lRUCache = new LRUCache(2);
lRUCache.put(1, 1); // cache is {1=1}
lRUCache.put(2, 2); // cache is {1=1, 2=2}
lRUCache.get(1);    // return 1
```

21.3K 253 586 Online

Testcase

Test Result

Accepted Runtime: 0 ms

Case 1

Input

["LRUCache","put","put","get","put","get","put","get","get","get"]

[[2],[1,1],[2,2],[1],[3,3],[2],[4,4],[1],[3],[4]]

Stdout

```
key=1
curr=a0
add key=1 val =1
key=2
curr=d0
add key=2 val =2
key=3
curr=100
```

View more

Output

[null,null,null,1,null,-1,null,-1,3,4]

Expected

[null,null,null,1,null,-1,null,-1,3,4]