Question: Implement own bind

Function.prototype.bind = function(context){

    const fn = this;

    const mainArguments = [].slice.call(arguments, 1);

    return function() {

        const currentArgs = [].slice.call(arguments);

        return fn.apply(context, [...mainArguments, ...currentArgs]);

    }

}

Question: Debounce

function debounce(func, interval, callfirst){

    let timeout;

    return function () {

        const context = this, args = arguments;

        let delay = function() {

            timeout = null;

            if(!callfirst){

            func.apply(context, args);

            }

        }

        const isCallFirst =  callfirst && !timeout;

        clearTimeout(timeout);

        timeout = setTimeout(delay, interval|| 500);

        if(isCallFirst) {

          func.apply(context, args);

        }

    }

}

Question : currying

function curry(func){

return  function (){

          const arguments1 = arguments;

        if(arguments1.length >= func.length){

            return func.apply(this, arguments1);

        }else{

        return function(){

             return curry.apply(this, func.bind(argument1.concat(arguments)))

         }

        }

    }

}

curriedSum(10, 20, 30) => 60

curriedSum(10, 20)(30) => 60

curriedSum(10)(20)(30) => 60

let curriedSum = curry(sum);

Question: implement loadsh get

\_\_get(object, keys, defaultVal = null): any {

keys = Array.isArray(keys) ? keys : keys.replace(/(\[(\d)\])/g, '.$2').split('.'); // split by dot of array

object = object[keys[0]];

if (object && keys.length > 1) {

return this.\_\_get(object, keys.slice(1), defaultVal);

}

return object === undefined ? defaultVal : object;

}

Question Link list

// adds an element at the end

// of list

add(element)

{

// creates a new node

var node = new Node(element);

// to store current node

var current;

// if list is Empty add the

// element and make it head

if (this.head == null)

this.head = node;

else {

current = this.head;

// iterate to the end of the

// list

while (current.next) {

current = current.next;

}

// add node

current.next = node;

}

this.size++;

}

// insert element at the position index

// of the list

insertAt(element, index)

{

if (index > 0 && index > this.size)

return false;

else {

// creates a new node

var node = new Node(element);

var curr, prev;

curr = this.head;

// add the element to the

// first index

if (index == 0) {

node.next = head;

this.head = node;

} else {

curr = this.head;

var it = 0;

// iterate over the list to find

// the position to insert

while (it < index) {

it++;

prev = curr;

curr = curr.next;

}

// adding an element

node.next = curr;

prev.next = node;

}

this.size++;

}

}

// removes an element from the

// specified location

removeFrom(index)

{

if (index > 0 && index > this.size)

return -1;

else {

var curr, prev, it = 0;

curr = this.head;

prev = curr;

// deleting first element

if (index == = 0) {

this.head = curr.next;

} else {

// iterate over the list to the

// position to removce an element

while (it < index) {

it++;

prev = curr;

curr = curr.next;

}

// remove the element

prev.next = curr.next;

}

this.size--;

// return the remove element

return curr.element;

}

}

// removes a given element from the

// list

removeElement(element)

{

var current = this.head;

var prev = null;

// iterate over the list

while (current != null) {

// comparing element with current

// element if found then remove the

// and return true

if (current.element == = element) {

if (prev == null) {

this.head = current.next;

} else {

prev.next = current.next;

}

this.size--;

return current.element;

}

prev = current;

current = current.next;

}

return -1;

}

// finds the index of element

indexOf(element)

{

var count = 0;

var current = this.head;

// iterae over the list

while (current != null) {

// compare each element of the list

// with given element

if (current.element == = element)

return count;

count++;

current = current.next;

}

// not found

return -1;

}

// gives the size of the list

size\_of\_list()

{

console.log(this.size);

}

class Node{

    constructor(data, next = null){

        this.data = data,

        this.next = next

    }

}

Question : memorization

const memoize = (callback, threshold = 1000) => {

let memo = new LRU(threshold);

return(args) => {

// console.log("get value", memo.get(args))

if(memo.get(args) != undefined){

console.log("from cache");

console.log(memo.getSizeOfCache())

return memo.get(args);

}else{

// const keys = Object.keys(memo);

// if(keys.length === threshold){

//     delete memo[keys[0]];

// }

 console.log(memo.getSizeOfCache())

memo.set(args, callback(args))

// memo[args] = callback(args);

// console.log("current size", keys.length);

return  memo.get(args);

}

}

}

Question : LRU

class LRU{

constructor(threshold = 100){

    this.max = threshold;

    this.cache = new Map();

}

get(key){

    // console.log("iun get", key)

    let item = this.cache.get(key);

     console.log(key,item)

    if(item != undefined){

        this.cache.delete(key);

        this.cache.set(key, item);

    }

   return item;

}

set(key, val){

    if(this.cache.has(key)){

         this.cache.delete(key);

    }

    if(this.cache.size === this.max){

        this.cache.delete(this.getFirst())

    }

    this.cache.set(key, val);

}

getFirst(){

    return this.cache.keys().next().value;

}

getSizeOfCache(){

    return this.cache.keys()

}

}

Question : chocolate/wrapper

static int countMaxChoco(int money,

                        int price, int wrap)

    {

        // Corner case

        if (money < price)

            return 0;

        // First find number of chocolates

        // that can be purchased with the

        // given amount

        int choc = money / price;

        // Now just add number of chocolates

        // with the chocolates gained by

        // wrapprices

        choc = choc + (choc - 1) / (wrap - 1);

        return choc;

    }

// recursion

int countRec(int choc, int wrap)

{

    // If number of chocolates is less than

    // number of wrappers required.

    if (choc < wrap)

        return 0;

    // We can immediatly get newChoc using

    // wrappers of choc.

    int newChoc = choc/wrap;

    // Now we have "newChoc + choc%wrap" wrappers.

    return newChoc + countRec(newChoc + choc%wrap,

                              wrap);

}

Question : maxDiff from an array

function getMaxiff (arr){

const length = arr.length;

let maxDiff = -100

if(!length){

return maxDiff;

}

let maximumRightValue = arr[length-1];

for(let i = length-2; i>=0 ; i--){

if(arr[i] > maximumRightValue){

maximumRightValue = arr[i];

}else{

const currentDiff = maximumRightValue - arr[i];

// console.log(currentDiff, arr[i])

if(currentDiff > maxDiff){

maxDiff = currentDiff;

}

}

}

return maxDiff;

}

console.log(getMaxiff([2, 3, 10, 6, 4, 8, 1]))

Question: Implement map function

const hoMyMap = function(callback, arr) {

    let resultArray = [];

    for(let i = 0; i<arr.length; i++){

        resultArray.push(callback(arr[i], i, this))

    }

    return resultArray;

}

Memorization for async

function memo(func, isAsync){

    let memo = {};

    const x = function(args){

        memo[args] = memo[args] || func.apply(this, args);

        return  memo[args];

    }

    x.store = memo;

    if(isAsync){

        return async function(){

            const args = JSON.stringify(arguments)

            x(arguments);

        }

    }else{

        return function(){

            const args =[].slice.call(arguments);

            x(args);

        }

    }

}

Memoization with cache return

function memo(func){

    let memo = {};

    let a = function(){

        const args = [].slice.call(arguments);

        if(memo[args]){

            return memo[args];

        }else{

            memo[args] = func.apply(this, args);

            return  memo[args];

        }

    };

    a.store = memo;

    return a;

}