# 单调栈及应用

## 一：单调栈

### 递增栈：

从栈底到栈顶是递增的的序列是递增的，如6，7，8，9，10

### 递减栈：

从栈底到栈顶是递减的，10，9，8，7，6.

### 总结：

用于求区间，非常实用。根本作用可以概括为，在一个一维数组中，帮助我们找到某个元素的左侧或右侧第一个比他大或者比他小的元素。具体的我们会维护一个栈，递增递减示情况而顶，这个栈满足一定情况会弹栈，一旦弹栈，必然就是出现比当前元素大或者小的元素。例如一组数：5，2，3，4，1

结论;

（1）求某元素右边第一个比他大的，维护递减栈（栈顶小）；左边第一个比他大的，维护递减栈；

解释：当nums[j] > stack[top]，【弹栈时更新】，说明nums[top]右边第一个比他大的就是nums[j];

当nums[j] < stack[top]，入栈时更新 说明nums[top]就是nums[j]左边第一个比他大的元素

（2）求某元素右边第一个比他小的，维护递增栈（栈顶元素大）；左边第一个比他小的，维护递增栈

解释：当nums[j] < stack[top],弹栈时更新，说明nums[top]的右边第一个比他小的就是nums[j]

当nums[j] > stack[top], 入栈时更新，说明nums[top]就是nums[j]左边第一个比他小的元素。

针对右边界关键理解弹栈的含义，针对左边界关键理解入栈的含义。

开胃菜：

nums[] = {5，2，3，4，1}

### （1）求每个元素右边第一个比他大的元素，输出数组target

int RightFirstBigger(int\* nums, int numsSize)

{

/\*\*维护递减栈,并且右边放置一个最大素，保证所有元素都弹出\*/

Stack sta = {0};

StackReInit(&sta);

int\* temp = (int\*)calloc(numsSize + 1, sizeof(int));

int\* target = (int\*)calloc(numsSize, sizeof(int));

int i;

int topVal;

memcpy(temp, nums, sizeof(int) \* (numsSize));

temp[numsSize] = INT\_MAX;

StackPush(&sta, 0);

for (i = 1; i < numsSize + 1; i++) {

topVal = StackTop(&sta);

if (temp[topVal] > temp[i]) {

StackPush(&sta, i);

continue;

}

while (StackIsEmpty(&sta) != 1 && temp[topVal] < temp[i]) {

topVal = StackPop(&sta);

/\*\*弹栈时temp[i]就是nums[topVal]右边第一个较大的元素\*/

target[topVal] = temp[i];

topVal = StackTop(&sta);

}

StackPush(&sta, i);

}

for (i = 0; i < numsSize; i++) {

printf("%d ", target[i]);

}

}

### （2）求每个元素左边第一个比他大的元素，输出数组target

int LeftFirstBigger(int\* nums, int numsSize)

{

/\*\*维护递减栈，保证所有元素都弹出\*/

Stack sta = {0};

StackReInit(&sta);

int\* temp = (int\*)calloc(numsSize + 1, sizeof(int));

int\* target = (int\*)calloc(numsSize, sizeof(int));

int i;

int topVal;

memcpy(temp, nums, sizeof(int) \* (numsSize));

StackPush(&sta, 0);

for (i = 1; i < numsSize + 1; i++) {

topVal = StackTop(&sta);

if (temp[topVal] > temp[i]) {

/\*\*入栈时temp[topVal]就是nums[i]左边第一个较大的元素\*/

target[i] = temp[topVal];

StackPush(&sta, i);

continue;

}

while (StackIsEmpty(&sta) != 1) {

StackPop(&sta);

topVal = StackTop(&sta);

if (temp[topVal] > temp[i]) {

break;

}

}

target[i] = temp[topVal];

StackPush(&sta, i);

}

for (i = 0; i < numsSize; i++) {

printf("%d ", target[i]);

}

}

### （3）求每个元素右边第一个比他小的元素，输出数组target

int RightFirstSmaller(int\* nums, int numsSize)

{

/\*\*维护递增栈,并且右边放置一个最小素，保证所有元素都弹出\*/

Stack sta = {0};

StackReInit(&sta);

int\* temp = (int\*)calloc(numsSize + 1, sizeof(int));

int\* target = (int\*)calloc(numsSize, sizeof(int));

int i;

int topVal;

memcpy(temp, nums, sizeof(int) \* (numsSize));

temp[numsSize] = INT\_MIN;

StackPush(&sta, 0);

for (i = 1; i < numsSize + 1; i++) {

topVal = StackTop(&sta);

if (temp[topVal] < temp[i]) {

StackPush(&sta, i);

continue;

}

while (StackIsEmpty(&sta) != 1) {

topVal = StackPop(&sta);

/\*\*temp[i]就是nums[topVal]右边第一个较小的元素\*/

target[topVal] = temp[i];

topVal = StackTop(&sta);

if (temp[topVal] < temp[i]) {

break;

}

}

StackPush(&sta, i);

}

for (i = 0; i < numsSize; i++) {

printf("%d ", target[i]);

}

}

### （4）求每个元素左边第一个比他小的元素，输出数组target

int LeftFirstSmaller(int\* nums, int numsSize)

{

/\*\*维护递增栈，保证所有元素都弹出\*/

Stack sta = {0};

StackReInit(&sta);

int\* temp = (int\*)calloc(numsSize + 1, sizeof(int));

int\* target = (int\*)calloc(numsSize, sizeof(int));

int i;

int topVal;

memcpy(temp, nums, sizeof(int) \* (numsSize));

StackPush(&sta, 0);

for (i = 1; i < numsSize + 1; i++) {

topVal = StackTop(&sta);

if (temp[topVal] < temp[i]) {

/\*\*temp[topVal]就是nums[i]左边第一个较大的元素\*/

target[i] = temp[topVal];

StackPush(&sta, i);

continue;

}

while (StackIsEmpty(&sta) != 1) {

StackPop(&sta);

topVal = StackTop(&sta);

if (topVal == -1 || temp[topVal] < temp[i]) {

break;

}

}

target[i] = topVal == -1 ? -1 : temp[topVal];

StackPush(&sta, i);

}

for (i = 0; i < numsSize; i++) {

printf("%d ", target[i]);

}

}

# 二：模板伪代码及分析

## 递增栈的维护.

for(遍历数组) {

while(！栈空) {

topVal = StackTop(&sta);// 获取栈顶元素

if (当前数组的元素 > 栈顶元素) {

入栈

继续

}

While (!栈空) {

topVal = StackTop(&sta);

if (source[i] > source[topVal]) {

target[topVal] = i - topVal;

StackPop(&sta);

continue;

}

StackPush(&sta, i);

break;

}

}

}

# 三：栈实现

#define MAX 100

typedef struct {

int top;

int size;

int pool[MAX];

}Stack;

void StackReInit(Stack\* sta)

{

sta->top = -1;

sta->size = 0;

memset(sta->pool, 0, MAX \* sizeof(int));

}

int StackIsEmpty(Stack\* sta)

{

return (sta->top) == -1 ? 1 : 0;

}

int StackIsFull(Stack\* sta)

{

return (sta->size == MAX) ? 1 : 0;

}

void StackPush(Stack\* sta, int value)

{

if (StackIsFull(sta) == 1) {

return;

}

sta->pool[++sta->top] = value;

sta->size++;

}

int StackPop(Stack\* sta)

{

if (StackIsEmpty(sta) == 1) {

return -1;

}

sta->size--;

return sta->pool[sta->top--];

}

int StackTop(Stack\* sta)

{

if (StackIsEmpty(sta) == 1) {

return -1;

}

return sta->pool[sta->top];

}

# 四：应用

## 2.视野总和

描叙：有n个人站队，所有的人全部向右看，个子高的可以看到个子低的发型，给出每个人的身高，问所有人能看到其他人发现总和是多少。  
输入：4 3 7 1  
输出：2  
解释：个子为4的可以看到个子为3的发型，个子为7可以看到个子为1的身高，所以1+1=2。

思路如下：即求每一个元素到其右边第一个比他大的元素的区间，然后相加。

int vis\_sum(int\* num, int size)

{

int\* stack = (int\*)malloc(sizeof(int) \* (size));

int\* v = (int\*)malloc(sizeof(int) \* (size + 1));

int sum = 0;

int i = 0, top = -1, temp = 0;

memset(stack, 0, sizeof(int) \* (size));

memset(v, 0, sizeof(int) \* (size + 1));

memcpy(v, num , sizeof(int) \* (size));

v[size] = 100;/ /理解为在最后插入一个最高的人，否则无法全部将有效数据弹出弹出

for (i = 0; i < size +1; i++) {

printf("%d ",v[i]);

}

for (i = 0; i < size + 1; i++) {

while (top != -1 && num[i] > num[stack[top]]) {

temp = stack[top];

top--;

sum += (i - temp - 1); }

top++;

stack[top] = i;

}

return sum;

}

int main()

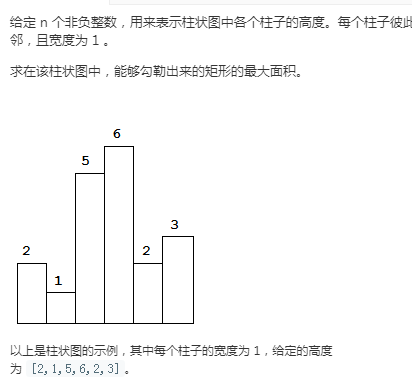
{

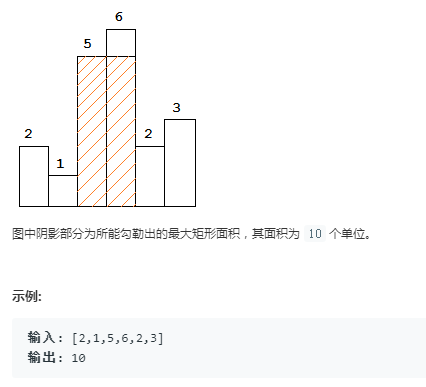
int num[] = {4, 3, 7, 1};

printf("\n%d\n",vis\_sum(num, 4));

}

## 4. [84. 柱状图中最大的矩形](https://leetcode-cn.com/problems/largest-rectangle-in-histogram/)





思路：

一个是高度（也即组成矩形的最短的那根柱子高度），一个是宽度，（也即组成矩形的柱子个数）。为了找到这个全局最大值，我们遍历所有局部最优情况。那么什么是局部最优解呢，我们将每个柱子的高度作为包含它的矩形的高度，也即这个柱子一定是这个矩形中最低的一个柱子，那么我们下一步是求解这个矩形的宽度，显然我们只需找到这个柱子左边，右边第一个比它低的柱子，就可以求出宽度。这显然让我们想到使用单调栈的数据结构。

#define GET\_MAX(x, y) ((x) > (y)) ? (x) : (y)

int largestRectangleArea(int\* heights, int heightsSize)

{

    int i;

    int\* heighTabel = (int\*)calloc(heightsSize + 1, sizeof(int));

    Stack stack = {0};

    int topVal;

    int maxArea = 0;

    int curArea;

    if (heightsSize == 0) {

        return 0;

    }

    if (heightsSize == 1) {

        return heights[0];

    }

    StackReInit(&stack);

    StackPush(&stack, 0);

    memcpy(heighTabel, heights, (heightsSize) \* sizeof(int));

    heighTabel[heightsSize] = 0;

    /\*\*其实就是寻找当前栈顶的左右第一个比他小的矩形宽度，因此递增栈。

    准备弹栈，说明碰到了右边第一小的，弹完栈之后的栈顶就是其左边第一小的元素\*/

    for (i = 1; i < heightsSize + 1; i++) {

        // 1.首先访问栈顶

        topVal = StackTop(&stack);

        // 2.如果比栈顶小，则一直弹栈直到非空

        while (StackIsEmpty(&stack) != 1 && heighTabel[i] <= heighTabel[topVal]) {

            // 3.取当前矩形的高度

            topVal = StackTop(&stack);

            // 弹栈下方公式i - StackTop(&stack) - 1为了求左边界

            StackPop(&stack);

            // 考虑到特殊情况，如果是弹出最后一个元素栈为空，此时的宽度是i

            curArea = heighTabel[topVal] \*((StackIsEmpty(&stack) == 1) ? (i) :(i - StackTop(&stack) - 1));

            maxArea = GET\_MAX(curArea, maxArea);

            // 4.弹出计算之后再次访问栈顶

            topVal = StackTop(&stack);

        }

        StackPush(&stack, i);

    }

    return maxArea;

}

#define MAX 100000

typedef struct {

    int top;

    int size;

    int pool[MAX];

}Stack;

void StackReInit(Stack\* sta)

{

    sta->top = -1;

    sta->size = 0;

    memset(sta->pool, 0, MAX \* sizeof(int));

}

int StackIsEmpty(Stack\* sta)

{

    return (sta->top) == -1 ? 1 : 0;

}

int StackIsFull(Stack\* sta)

{

    return (sta->size == MAX) ? 1 : 0;

}

void StackPush(Stack\* sta, int value)

{

    if (StackIsFull(sta) == 1) {

        return;

    }

    sta->pool[++sta->top] = value;

    sta->size++;

}

int StackPop(Stack\* sta)

{

    if (StackIsEmpty(sta) == 1) {

        return -1;

    }

    sta->size--;

    return sta->pool[sta->top--];

}

int StackTop(Stack\* sta)

{

    if (StackIsEmpty(sta) == 1) {

        return -1;

    }

    return sta->pool[sta->top];

}