C语言九日训练

第一天 函数

第一题 两整数之和(本题需要用到位运算)

给你两个整数 `a` 和 `b` ， **不使用** 运算符 `+` 和 `-` ，计算并返回两整数之和。

示例 1：

```
输入：a = 1, b = 2
输出：3
```

示例 2：

```
输入：a = 2, b = 3
输出：5
```

提示：

- `-1000 <= a, b <= 1000`

解答：

```c
int getSum(int a, int b) {
    while(a!=0)
    {
        unsigned int carry = (unsigned int) (a&b) << 1;
        int remain = a^b;
        a = carry;
        b = remain;
    }
    return b;
}
```

第二题 不用加法的加法

设计一个函数把两个数字相加。不得使用 + 或者其他算术运算符。

示例:

```
输入: a = 1, b = 1
输出: 2
```

提示：

- `a`, `b` 均可能是负数或 0
- 结果不会溢出 32 位整数

解答：

```
int add(int a, int b){
    while(a != 0)
    {
        unsigned int carry = (unsigned int)(a&b)<<1;
        int remain = a^b;
        a = carry;
        b = remain;
    }
    return b;
}
```

第三题 加密运算

计算机安全专家正在开发一款高度安全的加密通信软件，需要在进行数据传输时对数据进行加密和解密操作。假定 `dataA` 和 `dataB` 分别为随机抽样的两次通信的数据量：

- 正数为发送量
- 负数为接受量
- 0 为数据遗失

请不使用四则运算符的情况下实现一个函数计算两次通信的数据量之和（三种情况均需被统计），以确保在数据传输过程中的高安全性和保密性。

示例 1:

```
输入：dataA = 5, dataB = -1
输出：4
```

提示：

- `dataA` 和 `dataB` 均可能是负数或 0
- 结果不会溢出 32 位整数

解答：

```
int encryptionCalculate(int dataA, int dataB) {
    while(dataA!=0)
    {
        unsigned int carry = (unsigned int) (dataA&dataB) << 1;
        int remain = dataA^dataB;
        dataA = carry;
        dataB = remain;
    }
    return dataB;
}
```

第四题 递归乘法

递归乘法。 写一个递归函数，不使用 * 运算符， 实现两个正整数的相乘。可以使用加号、减号、位移，但要吝啬一些。

示例1:

```
输入：A = 1, B = 10
输出：10
```

示例2:

```
输入：A = 3, B = 4
输出：12
```

提示:

1. 保证乘法范围不会溢出

解答：

```
const int a = A;

if(B == 0)
{
    return 0;
}

else if(B > 0)
{
    while(B > 1)
    {
        B--;
        A += a;
    }
    return A;
}

else if(B < 0)
{
    while(B < -1){

    B++;
    A += a;
    }
    return -A;
}

return 0;
```

第五题 最大数值

编写一个方法，找出两个数字`a`和`b`中最大的那一个。不得使用if-else或其他比较运算符。

示例：

```
输入： a = 1, b = 2
输出： 2
```

解答：

```
int maximum(int a, int b){
    return a > b ? a : b;
}
```

位运算

与运算（&）

与运算符`&`对两个数的每一位进行与操作，只有当对应的两位都是1时，结果位才为1，否则为0。

或运算（|）

或运算符`|`对两个数的每一位进行或操作，只要对应的两位中有一个是1，结果位就为1。

异或运算（^）

异或运算符`^`对两个数的每一位进行异或操作，当对应的两位不同（一个为1，一个为0）时，结果位为1，否则为0。

取反运算（~）

取反运算符`~`对一个数的每一位进行取反操作，即将0变为1，1变为0。

左移运算（<<）

左移运算符`<<`将一个数的所有位向左移动指定的位数，右边用0填充。左移相当于乘以2的移动位数次方。

右移运算（>>）

右移运算符`>>`将一个数的所有位向右移动指定的位数，左边用符号位填充（对有符号数）或用0填充（对无符号数）。右移相当于除以2的移动位数次方（向下取整）。

消除n中最低位1：n & (n-1)

获取n中最低位：n & -n

交换代码

a= a^b;

b=a^b;

a = a^b;

（这是由异或的性质决定的。0^a = a; a^a = 0）

a+b可以转换为 「不带进位的加法运算」+ 「带进位的加法运算」

前者可以使用 a^b表示，后者使用 (a&b) << 1

第二天 循环

第一题 设计机械累加题

请设计一个机械累加器，计算从 1、2... 一直累加到目标数值 `target` 的总和。注意这是一个只能进行加法操作的程序，不具备乘除、if-else、switch-case、for 循环、while 循环，及条件判断语句等高级功能。

示例 1：

```
输入：target = 5
输出：15
```

示例 2：

```
输入：target = 7
输出：28
```

提示：

- `1 <= target <= 10000`

解答：

```
int mechanicalAccumulator(int target) {
    int a = 0;
    for(int i = 1;i <= target;i++)
    {
        a += i;
    }
    return a;
}
```

第二题 2的幂

给你一个整数 `n`，请你判断该整数是否是 2 的幂次方。如果是，返回 `true` ；否则，返回 `false` 。

如果存在一个整数 `x` 使得 `n == 2x` ，则认为 `n` 是 2 的幂次方。

示例 1：

```
输入：n = 1
输出：true
解释：20 = 1
```

示例 2：

```
输入：n = 16
输出：true
解释：24 = 16
```

示例 3：

```
输入：n = 3
输出：false
```

提示：

- `-231 <= n <= 231 - 1`

解答：

```
bool isPowerOfTwo(int n) {
    if(n <= 0)
    {
        return false;
    }
    for(int i = 1;i !=1073741824;i *= 2)
    {
        if(i == n)
        {
            return true;
        }
    }
    return n == 1073741824;
}
```

第三天 数组

第一题 斐波那契数

斐波那契数 （通常用 `F(n)` 表示）形成的序列称为 **斐波那契数列** 。该数列由 `0` 和 `1` 开始，后面的每一项数字都是前面两项数字的和。也就是：

```
F(0) = 0，F(1) = 1
F(n) = F(n - 1) + F(n - 2)，其中 n > 1
```

给定 `n` ，请计算 `F(n)` 。

示例 1：

```
输入：n = 2
输出：1
解释：F(2) = F(1) + F(0) = 1 + 0 = 1
```

示例 2：

```
输入：n = 3
输出：2
解释：F(3) = F(2) + F(1) = 1 + 1 = 2
```

示例 3：

```
输入：n = 4
输出：3
解释：F(4) = F(3) + F(2) = 2 + 1 = 3
```

提示：

- `0 <= n <= 30`

解答：

```
int fib(int n){
    if(n == 0)
    {
        return 0;
    }
    else if(n == 1)
    {
        return 1;
    }
    else
    {
        return fib(n-1)+fib(n-2);
    }
}
```

第二题 爬楼梯

假设你正在爬楼梯。需要 `n` 阶你才能到达楼顶。

每次你可以爬 `1` 或 `2` 个台阶。你有多少种不同的方法可以爬到楼顶呢？

示例 1：

```
输入：n = 2
输出：2
解释：有两种方法可以爬到楼顶。
1. 1 阶 + 1 阶
2. 2 阶
```

示例 2：

```
输入：n = 3
输出：3
解释：有三种方法可以爬到楼顶。
1. 1 阶 + 1 阶 + 1 阶
2. 1 阶 + 2 阶
3. 2 阶 + 1 阶
```

提示：

- `1 <= n <= 45`

解答：

```
int f[50];
int climbStairs(int n) {
    f[0] = 1;
    f[1] = 1;
    for(int i = 2;i <= n;++i)
    {
        f[i] = f[i-1] + f[i-2];
    }
    return f[n];
}
```

第三题 第N个泰波那锲数

泰波那契序列 Tn 定义如下：

T0 = 0, T1 = 1, T2 = 1, 且在 n >= 0 的条件下 Tn+3 = Tn + Tn+1 + Tn+2

给你整数 `n`，请返回第 n 个泰波那契数 Tn 的值。

示例 1：

```
输入：n = 4
输出：4
解释：
T_3 = 0 + 1 + 1 = 2
T_4 = 1 + 1 + 2 = 4
```

示例 2：

```
输入：n = 25
输出：1389537
```

提示：

- `0 <= n <= 37`
- 答案保证是一个 32 位整数，即 `answer <= 2^31 - 1`。

解答：

```
int f[40];
int tribonacci(int n){
    f[0] = 0;
    f[1] = 1;
    f[2] = 1;
    for(int i = 3;i <= n;++i)
    {
        f[i] = f[i-1] + f[i-2] + f[i-3];
    }
    return f[n];
}
```

第四题 差的绝对值为K的数对数目

给你一个整数数组 `nums` 和一个整数 `k` ，请你返回数对 `(i, j)` 的数目，满足 `i < j` 且 `|nums[i] - nums[j]| == k` 。

`|x|` 的值定义为：

- 如果 `x >= 0` ，那么值为 `x` 。
- 如果 `x < 0` ，那么值为 `-x` 。

示例 1：

```
输入：nums = [1,2,2,1], k = 1
输出：4
解释：差的绝对值为 1 的数对为：
- [1,2,2,1]
- [1,2,2,1]
- [1,2,2,1]
- [1,2,2,1]
```

示例 2：

```
输入：nums = [1,3], k = 3
输出：0
解释：没有任何数对差的绝对值为 3 。
```

示例 3：

```
输入：nums = [3,2,1,5,4], k = 2
输出：3
解释：差的绝对值为 2 的数对为：
- [3,2,1,5,4]
- [3,2,1,5,4]
- [3,2,1,5,4]
```

提示：

- `1 <= nums.length <= 200`

- `1 <= nums[i] <= 100`
- `1 <= k <= 99`

解答：

```
int countKDifference(int* nums, int numsSize, int k) {
    int a = 0;
    for(int i = 0;i < numsSize;i++)
    {
        for(int j = i+1;j < numsSize;j++)
        {
            if(abs(nums[i]-nums[j])==k)a++;
        }
    }
    return a;
}
```

第五题 拿硬币(贪心)

桌上有 `n` 堆力扣币，每堆的数量保存在数组 `coins` 中。我们每次可以选择任意一堆，拿走其中的一枚或者两枚，求拿完所有力扣币的最少次数。

示例 1：

> 输入：`[4,2,1]`
>
> 输出：`4`
>
> 解释：第一堆力扣币最少需要拿 2 次，第二堆最少需要拿 1 次，第三堆最少需要拿 1 次，总共 4 次即可拿完。

示例 2：

> 输入：`[2,3,10]`
>
> 输出：`8`

限制：

- `1 <= n <= 4`
- `1 <= coins[i] <= 10`

解答：

```
int minCount(int* coins, int coinsSize){
    int ans = 0;
    for(int i = 0;i < coinsSize;i++)
    {
        if(coins[i]%2 != 0)
        {
            ans += (coins[i]+1)/2;
        }
        else
        {
            ans += coins[i]/2;
        }
    }
```

```
    return ans;
}
```

动态规划

动态规划（Dynamic Programming,DP）算法通常用于求解某种具有最优性质的问题。在这类问题中，可能会有许多可行解，每一个解都对应一个值，我们希望找到具有最优值的解。
动态规划算法与分治法类似，其基本思想也是将待求解的问题分解成若干个子问题，先求解子问题，然后从这些子问题的解中 得到原有问题的解。与分治法不同的是，动态规划经分解后得到的子问题往往 不是相互独立 的。
这里只做了解，后续在数据结构和算法中继续学习

第四天 指针

第一题 重新排列数列

给你一个数组 `nums` ，数组中有 `2n` 个元素，按 `[x1,x2,...,xn,y1,y2,...,yn]` 的格式排列。

请你将数组按 `[x1,y1,x2,y2,...,xn,yn]` 格式重新排列，返回重排后的数组。

示例 1：

```
输入：nums = [2,5,1,3,4,7], n = 3
输出：[2,3,5,4,1,7]
解释：由于 x1=2, x2=5, x3=1, y1=3, y2=4, y3=7 ，所以答案为 [2,3,5,4,1,7]
```

示例 2：

```
输入：nums = [1,2,3,4,4,3,2,1], n = 4
输出：[1,4,2,3,3,2,4,1]
```

示例 3：

```
输入：nums = [1,1,2,2], n = 2
输出：[1,2,1,2]
```

解答：

```
int* shuffle(int* nums, int numsSize, int n, int* returnSize){
    int *ans = (int *)malloc(sizeof(int) * n * 2);
    for (int i = 0; i < n; i++) {
        ans[2 * i] = nums[i];
        ans[2 * i + 1] = nums[i + n];
    }
    *returnSize = n * 2;
    return ans;
}
```

第二题 数组串联

给你一个长度为 `n` 的整数数组 `nums` 。请你构建一个长度为 `2n` 的答案数组 `ans` ，数组下标 **从 0 开始计数** ，对于所有 `0 <= i < n` 的 `i` ，满足下述所有要求：

- `ans[i] == nums[i]`
- `ans[i + n] == nums[i]`

具体而言，`ans` 由两个 `nums` 数组 **串联** 形成。

返回数组 `ans` 。

示例 1：

```
输入：nums = [1,2,1]
输出：[1,2,1,1,2,1]
解释：数组 ans 按下述方式形成：
- ans = [nums[0],nums[1],nums[2],nums[0],nums[1],nums[2]]
- ans = [1,2,1,1,2,1]
```

示例 2：

```
输入：nums = [1,3,2,1]
输出：[1,3,2,1,1,3,2,1]
解释：数组 ans 按下述方式形成：
- ans = [nums[0],nums[1],nums[2],nums[3],nums[0],nums[1],nums[2],nums[3]]
- ans = [1,3,2,1,1,3,2,1]
```

提示：

- `n == nums.length`
- `1 <= n <= 1000`
- `1 <= nums[i] <= 1000`

解答：

```c
int* getConcatenation(int* nums, int numsSize, int* returnSize) {
    int* ans = (int*)malloc(sizeof(int)*numsSize*2);
    for(int i=0;i < numsSize;i++)
    {
        ans[i]=nums[i];
    }
    for(int i=0;i < numsSize;i++)
    {
        ans[i+numsSize]=nums[i];
    }
    *returnSize = numsSize*2;
    return ans;
}
```

第三题 一维数组的动态和

给你一个数组 `nums` 。数组「动态和」的计算公式为：`runningSum[i] = sum(nums[0]…nums[i])` 。

请返回 `nums` 的动态和。

示例 1：

```
输入：nums = [1,2,3,4]
输出：[1,3,6,10]
解释：动态和计算过程为 [1, 1+2, 1+2+3, 1+2+3+4] 。
```

示例 2：

```
输入：nums = [1,1,1,1,1]
输出：[1,2,3,4,5]
解释：动态和计算过程为 [1, 1+1, 1+1+1, 1+1+1+1, 1+1+1+1+1] 。
```

示例 3：

```
输入：nums = [3,1,2,10,1]
输出：[3,4,6,16,17]
```

提示：

- `1 <= nums.length <= 1000`
- `-10^6 <= nums[i] <= 10^6`

解答：

```
int* runningSum(int* nums, int numsSize, int* returnSize){
    int* ans = (int*)malloc(sizeof(int)*numsSize);
    *returnSize = numsSize;
    for(int i = 0;i < numsSize;i++)
    {
        int result = 0;
        for(int j = 0;j <= i;j++)
        {
            if(i == 0)
            {
                result = nums[i];
            }
            else
            {
                result += nums[j];
            }
            ans[j] = result;
        }
    }
    return ans;
}
```

第四题 动态口令

某公司门禁密码使用动态口令技术。初始密码为字符串 `password`，密码更新均遵循以下步骤：

- 设定一个正整数目标值 `target`
- 将 `password` 前 `target` 个字符按原顺序移动至字符串末尾

请返回更新后的密码字符串。

示例 1：

```
输入：password = "s3cur1tyC0d3", target = 4
输出："r1tyC0d3s3cu"
```

示例 2：

```
输入：password = "lrloseumgh", target = 6
输出："umghlrlose"
```

提示：

- `1 <= target < password.length <= 10000`

解答：

```
char* dynamicPassword(char* password, int target) {
    int n = strlen(password);
    char* new_password = (char*)malloc(sizeof(char)*(n+1));
    for(int i = 0;i < n;++i)
    {
        new_password[i] = password[(i+target)%n];
    }
    new_password[n] = '\0';
    return new_password;
}
```

第五天 排序

这里的排序使用的是qsort这里的排序使用的是qsort

第一题 排序数组

给你一个整数数组 `nums`，请你将该数组升序排列。

示例 1：

```
输入：nums = [5,2,3,1]
输出：[1,2,3,5]
```

示例 2：

```
输入：nums = [5,1,1,2,0,0]
输出：[0,0,1,1,2,5]
```

提示：

- `1 <= nums.length <= 5 * 104`
- `-5 * 104 <= nums[i] <= 5 * 104`

解答：

```
int cmp(const void *a,const void *b)
{
    int vala = *(int*)a;
    int valb = *(int*)b;
    return vala > valb ? 1:-1;
}
int* sortArray(int* nums, int numsSize, int* returnSize) {
    qsort(nums,numsSize,sizeof(int),cmp);
    *returnSize = numsSize;
    return nums;
}
```

第六天 贪心

第一题 两个数对之间的最大乘积差

两个数对 `(a, b)` 和 `(c, d)` 之间的 **乘积差** 定义为 `(a * b) - (c * d)` 。

- 例如，`(5, 6)` 和 `(2, 7)` 之间的乘积差是 `(5 * 6) - (2 * 7) = 16` 。

给你一个整数数组 `nums` ，选出四个 **不同的** 下标 `w`、`x`、`y` 和 `z` ，使数对 `(nums[w], nums[x])` 和 `(nums[y], nums[z])` 之间的 **乘积差** 取到 **最大值** 。

返回以这种方式取得的乘积差中的 **最大值** 。

示例 1：

```
输入：nums = [5,6,2,7,4]
输出：34
解释：可以选出下标为 1 和 3 的元素构成第一个数对 (6, 7) 以及下标 2 和 4 构成第二个数对 (2, 4)
乘积差是 (6 * 7) - (2 * 4) = 34
```

示例 2：

```
输入：nums = [4,2,5,9,7,4,8]
输出：64
解释：可以选出下标为 3 和 6 的元素构成第一个数对 (9, 8) 以及下标 1 和 5 构成第二个数对 (2, 4)
乘积差是 (9 * 8) - (2 * 4) = 64
```

提示：

- `4 <= nums.length <= 104`
- `1 <= nums[i] <= 104`

解答：

```
int cmp(const void *a,const void *b)
{
    int vala = *(int*)a;
    int valb = *(int*)b;
    return vala > valb ? 1:-1;
}
int maxProductDifference(int* nums, int numsSize){
    qsort(nums,numsSize,sizeof(int),cmp);
    return nums[numsSize-1]*nums[numsSize-2] - nums[0]*nums[1];
}
```

第二题 三角形的最大周长

给定由一些正数（代表长度）组成的数组 `nums` ，返回 *由其中三个长度组成的、**面积不为零**的三角形的最大周长* 。如果不能形成任何面积不为零的三角形，返回 `0` 。

示例 1：

```
输入：nums = [2,1,2]
输出：5
解释：你可以用三个边长组成一个三角形:1 2 2。
```

示例 2：

```
输入：nums = [1,2,1,10]
输出：0
解释：
你不能用边长 1,1,2 来组成三角形。
不能用边长 1,1,10 来构成三角形。
不能用边长 1、2 和 10 来构成三角形。
因为我们不能用任何三条边长来构成一个非零面积的三角形，所以我们返回 0。
```

提示：

- `3 <= nums.length <= 104`
- `1 <= nums[i] <= 106`

解答：

```
int cmp(const void *a,const void *b)
{
```

```
    int vala = *(int*)a;
    int valb = *(int*)b;
    return vala > valb ? 1:-1;
}
int largestPerimeter(int* nums, int numsSize) {
    qsort(nums,numsSize,sizeof(int),cmp);
    for(int i = numsSize-1;i >= 2;--i)
    {
        int c = nums[i];
        int a = nums[i-1];
        int b = nums[i-2];
        if(a+b > c)
        {
            return a+b+c;
        }
    }
    return 0;
}
```

第三题 数值拆分

给定长度为 `2n` 的整数数组 `nums` ，你的任务是将这些数分成 `n` 对, 例如 `(a1, b1), (a2, b2), ..., (an, bn)` ，使得从 `1` 到 `n` 的 `min(ai, bi)` 总和最大。

返回该 **最大总和** 。

示例 1：

```
输入：nums = [1,4,3,2]
输出：4
解释：所有可能的分法（忽略元素顺序）为：
1. (1, 4), (2, 3) -> min(1, 4) + min(2, 3) = 1 + 2 = 3
2. (1, 3), (2, 4) -> min(1, 3) + min(2, 4) = 1 + 2 = 3
3. (1, 2), (3, 4) -> min(1, 2) + min(3, 4) = 1 + 3 = 4
所以最大总和为 4
```

示例 2：

```
输入：nums = [6,2,6,5,1,2]
输出：9
解释：最优的分法为 (2, 1), (2, 5), (6, 6). min(2, 1) + min(2, 5) + min(6, 6) = 1 + 2
+ 6 = 9
```

提示：

- `1 <= n <= 104`
- `nums.length == 2 * n`
- `-104 <= nums[i] <= 104`

解答：

```
int cmp(const void *a,const void *b)
{
    int vala = *(int*)a;
    int valb = *(int*)b;
    return vala > valb ? 1:-1;
}
int arrayPairSum(int* nums, int numsSize) {
    qsort(nums,numsSize,sizeof(int),cmp);
    int ans = 0;
    for(int i = 0;i < numsSize;i += 2)
    {
        ans += nums[i];
    }
    return ans;
}
```

第四题 救生艇

给定数组 `people` 。`people[i]`表示第 `i` 个人的体重 ，**船的数量不限**，每艘船可以承载的最大重量为 `limit`。

每艘船最多可同时载两人，但条件是这些人的重量之和最多为 `limit`。

返回 *承载所有人所需的最小船数* 。

示例 1：

```
输入：people = [1,2], limit = 3
输出：1
解释：1 艘船载 (1, 2)
```

示例 2：

```
输入：people = [3,2,2,1], limit = 3
输出：3
解释：3 艘船分别载 (1, 2), (2) 和 (3)
```

示例 3：

```
输入：people = [3,5,3,4], limit = 5
输出：4
解释：4 艘船分别载 (3), (3), (4), (5)
```

提示：

- `1 <= people.length <= 5 * 104`
- `1 <= people[i] <= limit <= 3 * 104`

解答：

```
int cmp(const void *a,const void *b)
```

```
{
    int vala = *(int*)a;
    int valb = *(int*)b;
    return vala > valb ? 1:-1;
}
int numRescueBoats(int* people, int peopleSize, int limit) {
    qsort(people,peopleSize,sizeof(int),cmp);
    int l = 0,r = peopleSize-1;
    int ans = 0;
    while(l <= r)
    {
        if(l == r)
        {
            ans += 1;
            return ans;
        }
        else
        {
            if(people[l]+people[r]<=limit)
            {
                l++;
                r--;
                ans += 1;
            }
            else{
                r--;
                ans += 1;
            }
        }
    }
    return ans;
}
```

贪心算法的概念

贪心算法（Greedy Alogorithm）又叫登山算法，它的根本思想是逐步到达山顶，即逐步获得最优解，是解决最优化问题时的一种简单但是适用范围有限的策略。

贪心算法没有固定的框架，算法设计的关键是贪婪策略的选择。贪心策略要无后向性，也就是说某状态以后的过程不会影响以前的状态，至于当前状态有关。

贪心算法是对某些求解最优解问题的最简单、最迅速的技术。某些问题的最优解可以通过一系列的最优的选择即贪心选择来达到。但局部最优并不总能获得整体最优解，但通常能获得近似最优解。

在每一步贪心选择中，只考虑当前对自己最有利的选择，而不去考虑在后面看来这种选择是否合理。

第七天 二维数组

第一题 统计有序矩阵中的负数

给你一个 `m * n` 的矩阵 `grid`，矩阵中的元素无论是按行还是按列，都以非严格递减顺序排列。 请你统计并返回 `grid` 中 **负数** 的数目。

示例 1：

```
输入：grid = [[4,3,2,-1],[3,2,1,-1],[1,1,-1,-2],[-1,-1,-2,-3]]
输出：8
解释：矩阵中共有 8 个负数。
```

示例 2：

```
输入：grid = [[3,2],[1,0]]
输出：0
```

提示：

- `m == grid.length`
- `n == grid[i].length`
- `1 <= m, n <= 100`
- `-100 <= grid[i][j] <= 100`

解答：

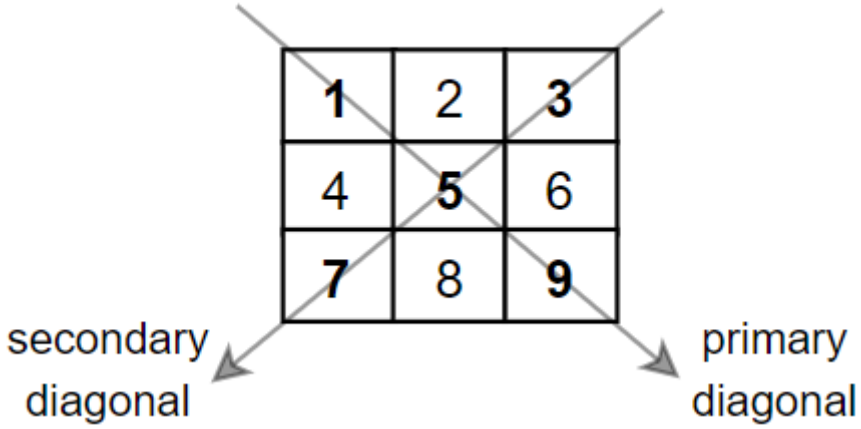
```
int countNegatives(int** grid, int gridSize, int* gridColSize) {
    int r = gridSize;
    int c = gridColSize[0];
    int ans = 0;
    for(int i = 0;i < r;++i)
    {
        for(int j = 0;j < c;++j)
        {
            if(grid[i][j] < 0)++ans;
        }
    }
    return ans;
}
```

第二题 矩阵对角线元素的和

给你一个正方形矩阵 `mat`，请你返回矩阵对角线元素的和。

请你返回在矩阵主对角线上的元素和副对角线上且不在主对角线上元素的和。

示例 1：




```
输入：mat = [[1,2,3],
             [4,5,6],
             [7,8,9]]
输出：25
解释：对角线的和为：1 + 5 + 9 + 3 + 7 = 25
请注意，元素 mat[1][1] = 5 只会被计算一次。
```

示例 2：

```
输入：mat = [[1,1,1,1],
             [1,1,1,1],
             [1,1,1,1],
             [1,1,1,1]]
输出：8
```

示例 3：

```
输入：mat = [[5]]
输出：5
```

提示：

- `n == mat.length == mat[i].length`
- `1 <= n <= 100`
- `1 <= mat[i][j] <= 100`

解答：

```c
int diagonalSum(int** mat, int matSize, int* matColSize) {
    int n = matSize;
    int ans = 0;
    for(int i = 0;i < n;++i)
    {
        ans += mat[i][i];
    }
    for(int i = 0;i < n;++i)
        ans += mat[i][n-1-i];
    if(n % 2)
    {
```

```
        ans -= mat[n/2][n/2];
    }
    return ans;
}
```

第八天 二级指针

第一题 翻转图像

给定一个 `n x n` 的二进制矩阵 `image` ，先 **水平** 翻转图像，然后 **反转** 图像并返回 *结果* 。

水平翻转图片就是将图片的每一行都进行翻转，即逆序。

- 例如，水平翻转 `[1,1,0]` 的结果是 `[0,1,1]`。

反转图片的意思是图片中的 `0` 全部被 `1` 替换， `1` 全部被 `0` 替换。

- 例如，反转 `[0,1,1]` 的结果是 `[1,0,0]`。

示例 1：

```
输入：image = [[1,1,0],[1,0,1],[0,0,0]]
输出：[[1,0,0],[0,1,0],[1,1,1]]
解释：首先翻转每一行: [[0,1,1],[1,0,1],[0,0,0]]；
      然后反转图片: [[1,0,0],[0,1,0],[1,1,1]]
```

示例 2：

```
输入：image = [[1,1,0,0],[1,0,0,1],[0,1,1,1],[1,0,1,0]]
输出：[[1,1,0,0],[0,1,1,0],[0,0,0,1],[1,0,1,0]]
解释：首先翻转每一行: [[0,0,1,1],[1,0,0,1],[1,1,1,0],[0,1,0,1]]；
      然后反转图片: [[1,1,0,0],[0,1,1,0],[0,0,0,1],[1,0,1,0]]
```

提示：

- `n == image.length`
- `n == image[i].length`
- `1 <= n <= 20`
- `images[i][j]` == `0` 或 `1`.

解答：

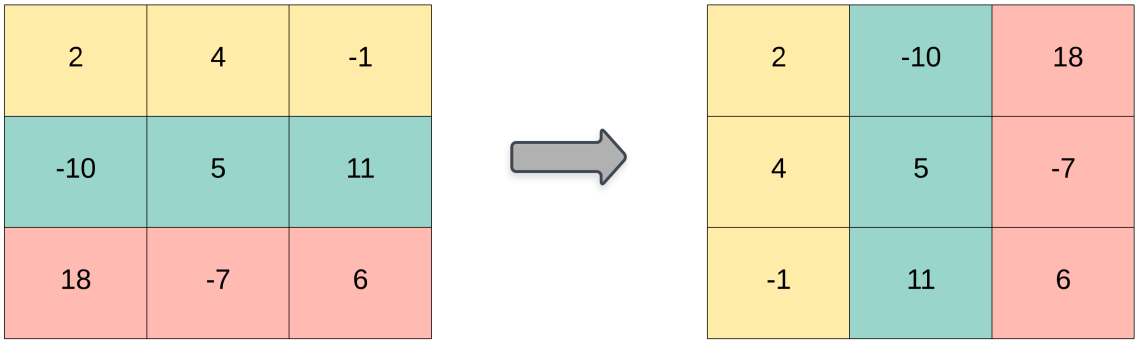
```
int** flipAndInvertImage(int** image, int imageSize, int* imageColSize, int*
returnSize, int** returnColumnSizes) {
    *returnSize = imageSize;
    *returnColumnSizes = imageColSize;
    int n = imageSize;
    for (int i = 0; i < n; i++) {
        int left = 0, right = n - 1;
        while (left < right) {
```

```
            if (image[i][left] == image[i][right]) {
                image[i][left] ^= 1;
                image[i][right] ^= 1;
            }
            left++;
            right--;
        }
        if (left == right) {
            image[i][left] ^= 1;
        }
    }
    return image;
}
```

第二题 转置矩阵

给你一个二维整数数组 `matrix`， 返回 `matrix` 的 **转置矩阵** 。

矩阵的 **转置** 是指将矩阵的主对角线翻转，交换矩阵的行索引与列索引。



示例 1：

```
输入：matrix = [[1,2,3],[4,5,6],[7,8,9]]
输出：[[1,4,7],[2,5,8],[3,6,9]]
```

示例 2：

```
输入：matrix = [[1,2,3],[4,5,6]]
输出：[[1,4],[2,5],[3,6]]
```

提示：

- `m == matrix.length`
- `n == matrix[i].length`
- `1 <= m, n <= 1000`
- `1 <= m * n <= 105`
- `-109 <= matrix[i][j] <= 109`

解答：

```c
int** transpose(int** matrix, int matrixSize, int* matrixColSize, int*
returnSize, int** returnColumnSizes) {
    *returnSize = matrixColSize[0];
    int m = matrixSize;
    int n = matrixColSize[0];
    int **res = (int**)malloc(sizeof(int*)*n);
    *returnColumnSizes = (int*)malloc(sizeof(int) * n);
    for(int i = 0;i < n;i++)
    {
        res[i] = (int*)malloc(sizeof(int)*m);
        (*returnColumnSizes)[i] = m;
        for(int j = 0;j < m;j++)
        {
            res[i][j] = matrix[j][i];
        }
    }
    return res;
}
```

第三题 重塑矩阵

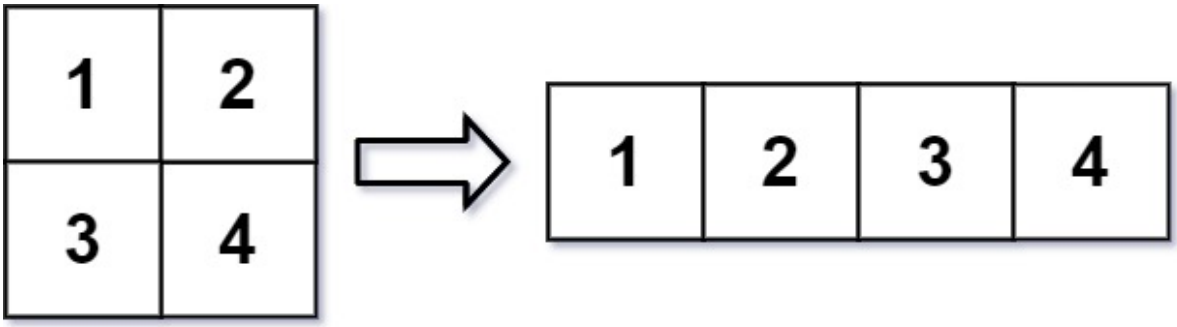
在 MATLAB 中，有一个非常有用的函数 `reshape` ，它可以将一个 `m x n` 矩阵重塑为另一个大小不同（`r x c`）的新矩阵，但保留其原始数据。

给你一个由二维数组 `mat` 表示的 `m x n` 矩阵，以及两个正整数 `r` 和 `c` ，分别表示想要的重构的矩阵的行数和列数。

重构后的矩阵需要将原始矩阵的所有元素以相同的 **行遍历顺序** 填充。

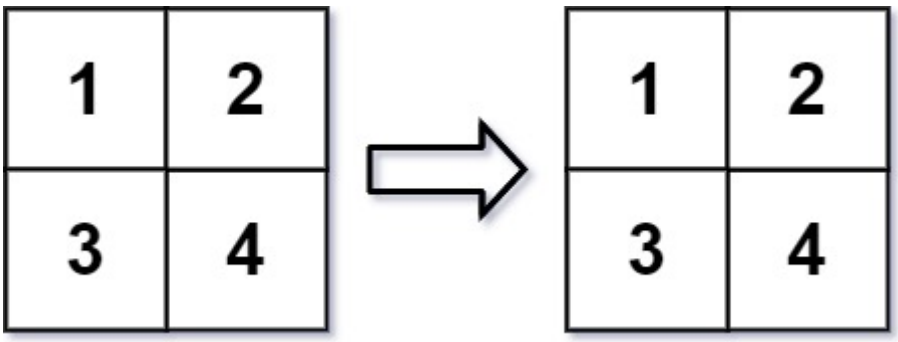
如果具有给定参数的 `reshape` 操作是可行且合理的，则输出新的重塑矩阵；否则，输出原始矩阵。

示例 1：




```
输入：mat = [[1,2],[3,4]], r = 1, c = 4
输出：[[1,2,3,4]]
```

示例 2：




```
输入：mat = [[1,2],[3,4]], r = 2, c = 4
输出：[[1,2],[3,4]]
```

提示：

- `m == mat.length`
- `n == mat[i].length`
- `1 <= m, n <= 100`
- `-1000 <= mat[i][j] <= 1000`
- `1 <= r, c <= 300`

解答：

```
/**
 * Return an array of arrays of size *returnSize.
 * The sizes of the arrays are returned as *returnColumnSizes array.
 * Note: Both returned array and *columnSizes array must be malloced, assume
caller calls free().
 */
int** matrixReshape(int** mat, int matSize, int* matColSize, int r, int c, int*
returnSize, int** returnColumnSizes) {
    int n = matSize;       // 原始矩阵的行数
    int m = matColSize[0]; // 原始矩阵的列数

    // 检查是否可以进行重塑
    if (m * n != r * c) {
        *returnSize = n;
        *returnColumnSizes = (int*)malloc(n * sizeof(int));
        for (int i = 0; i < n; i++) {
            (*returnColumnSizes)[i] = m;
        }
        return mat; // 不能重塑时，返回原始矩阵
    }

    // 分配内存
    int** res = (int**)malloc(r * sizeof(int*));
    *returnColumnSizes = (int*)malloc(r * sizeof(int));
    for (int i = 0; i < r; i++) {
        res[i] = (int*)malloc(c * sizeof(int));
        (*returnColumnSizes)[i] = c;
    }

    // 填充新矩阵
```

```
    for (int i = 0; i < r * c; i++) {
        res[i / c][i % c] = mat[i / m][i % m];
    }

    *returnSize = r;
    return res;
}
```

第九天 递归

第一题 阶乘后的零

给定一个整数 `n` ，返回 `n!` 结果中尾随零的数量。

提示 `n! = n * (n - 1) * (n - 2) * ... * 3 * 2 * 1`

示例 1：

```
输入：n = 3
输出：0
解释：3! = 6 ，不含尾随 0
```

示例 2：

```
输入：n = 5
输出：1
解释：5! = 120 ，有一个尾随 0
```

示例 3：

```
输入：n = 0
输出：0
```

提示：

- `0 <= n <= 104`

解答：

```
int trailingZeroes(int n){
    if(n < 5){
        return 0;
    }
    int temp = n/5 + trailingZeroes(n/5);
    return temp;
}
```

第二题 将数字变成0的操作次数

给你一个非负整数 `num` ，请你返回将它变成 0 所需要的步数。 如果当前数字是偶数，你需要把它除以 2 ；否则，减去 1 。

示例 1:

```
输入：num = 14
输出：6
解释：
步骤 1) 14 是偶数，除以 2 得到 7 。
步骤 2） 7 是奇数，减 1 得到 6 。
步骤 3） 6 是偶数，除以 2 得到 3 。
步骤 4） 3 是奇数，减 1 得到 2 。
步骤 5） 2 是偶数，除以 2 得到 1 。
步骤 6） 1 是奇数，减 1 得到 0 。
```

示例 2:

```
输入：num = 8
输出：4
解释：
步骤 1） 8 是偶数，除以 2 得到 4 。
步骤 2） 4 是偶数，除以 2 得到 2 。
步骤 3） 2 是偶数，除以 2 得到 1 。
步骤 4） 1 是奇数，减 1 得到 0 。
```

示例 3:

```
输入：num = 123
输出：12
```

提示:

- `0 <= num <= 10^6`

解答:

```c
int numberOfSteps(int num) {
    if(num == 0)
    {
        return 0;
    }
    if(num%2)
    {
        return numberOfSteps(num-1) + 1;
    }
    return numberOfSteps(num/2)+1;
}
```