线索 二叉树

```
// 线索/孩子
#define THREAD 1
#define CHILD 0

typedef struct TreeNode {
  int val;
  struct TreeNode *left;
  struct TreeNode *right;
  int leftTag,rightTag;
}TreeNode;

// 中序创建线索二叉树，参数是根节点指针和前驱节点
void inThread(TreeNode *root,TreeNode *pre){
    if(root){
        inThread(root->left,pre);
        if(!root->left){
            root->left = pre;
            root->leftTag = THREAD;
        }

        if(pre && !pre->right){
            pre->right = root;
            pre->rightTag = THREAD;
        }
        pre = root;
        inThread(root->right,pre);
    }
}

// 获取当前根节点对应树的中序遍历第一个节点
TreeNode* getFirstNode(TreeNode *root){
    TreeNode *temp = root;
    // 一直到最左边节点
    while(temp && temp->leftTag == CHILD){
        temp = temp->left;
    }
    return temp;
}

// 获取当前节点中序遍历的下一个节点
TreeNode* nextNode(TreeNode *root){
```

```
    if(root == NULL){
        return NULL;
    }
    //  当前节点有右孩子，则需要访问到右孩子树的最左边节点
    if(root->rightTag == CHILD){
        return getFirstNode(root->right);
    }
    // 如果当前节点有右线索，直接返回右线索即可
    else{
        return root->right;
    }
}

// 中序遍历线索二叉树
void inTraverse(TreeNode *root){
    for(TreeNode *p = getFirstNode(root);p!=NULL;p = nextNode(p)){
        printf("%d\n",p->val);
    }
}
```

并查集

```c
#include <stdio.h>
#define SIZE 10


// 保存每个顶点的父节点
static int parent[SIZE] = {0};

// 保存当前节点的rank,防止合并时出现单链的情况
static int rank[SIZE] = {0};

// 初始时所有的节点构成单独的集合，父节点为-1
void initialize(){
    for(int i = 0;i <  SIZE;i++){
        parent[i] = -1;
    }
}

// 寻找x节点所在的集合树的根节点
int findRoot(int x){
    int temp = x;
    while(parent[temp] != -1){
        temp = parent[temp];
    }
    return temp;
}

// 将两个集合元素合并到一个集合，返回1表示两个元素不在同一个集合，可以合并
// 返回0表示两个集合在同一个集合，合并失败
int doUnion(int x,int y){
    int x_root =  findRoot(x);
    int y_root = findRoot(y);
    if(x_root == y_root){
        return 0;
    }

    // 将rank小的根节点挂到rank大的节点的下面
    if(rank[x_root] < rank[y_root]){
        rank[x_root] = y_root;
    }else if(rank[x_root] > rank[y_root]){
        rank[y_root] = x_root;
```

```
    }else{
        rank[y_root]++;
        rank[x_root] = y_root;
    }
    return 1;
}

int main(){
    int edges[][2] = {{0,1},{1,2},{3,5},{3,7},{5,6},{6,8},
    {7,8},{8,9},{6,9}};
    int length = sizeof(edges) / (sizeof(int) * 2);

    initialize();

    for(int i = 0;i < length;i++){
        doUnion(edges[i][0],edges[i][1]);
    }
    for(int j = 0;j < SIZE;j++){
        printf("%d根节点:%d\n",j,findRoot(j));
    }
    return 0;
}
```

KMP算法

```c
#include <stdio.h>
#include <stdlib.h>
void getNext(char *pattern, int *next, int size)
{

    // i指向当前待求位置，j始终指向模式串中匹配的最长前后缀
    // 前缀的下一个位置
    int i = 0, j = -1;
    next[0] = -1;

    while (i < size - 1)
    {
        // j = -1时说明模式串没有匹配的前后缀，从第一个位置开始
        // 如果 pattern[i] == pattern[j] 说明最长匹配前后缀
        // 加上当前字符后依然构成匹配的前后缀
        if (j == -1 || pattern[i] == pattern[j])
        {
            next[++i] = ++j;
        }
        else
        // 如果不匹配则寻找上一个最长匹配前缀的下一个位置
        {
            j = next[j];
        }
    }
}

int kmp(char *str, char *pattern)
{

    if (!str || !pattern)
        return -1;
    int len;
    for (len = 0; pattern[len] != '\0'; len++)
    {
    }

    int *next = (int *)malloc(sizeof(int) * len);

    getNext(pattern, next, len);
```

```
    int i = 0, j = 0;
    while (str[i] != '\0' && pattern[j] != '\0')
    {

        if (j == -1 || pattern[j] == str[i])
        {
            i++;
            j++;
        }
        else
        {
            j = next[j];
        }
    }

    if (pattern[j] == '\0')
    {
        return i - j;
    }
    else
    {
        return -1;
    }
}
```

构建平衡二叉树

```
// 将一个按照升序排列的有序数组，转换为一棵高度平衡二叉搜索树。

// 本题中，一个高度平衡二叉树是指一个二叉树每个节点 的左右两个子树的高度差的绝对值不超过 1。

typedef struct TreeNode {
  int val;
  struct TreeNode *left;
  struct TreeNode *right;
}TreeNode;

TreeNode* toTree(int arr[],int left,int right){
  if(left <= right){
    int mid = (left + right) / 2;
    TreeNode *root = (TreeNode*)malloc(sizeof(TreeNode));
    root->val = arr[mid];
    root->left = toTree(arr,left,mid - 1);
    root->right = toTree(arr,mid + 1,right);
    return root;
  }else{
    return NULL;
  }
}

struct TreeNode* sortedArrayToBST(int* nums, int numsSize){
  if(nums == NULL || numsSize <= 0){
    return NULL;
  }
  return toTree(nums,0,numsSize - 1);
}
```

设计最小栈

```c
/** 设计一个支持 push，pop，top 操作，并能在常数时间内检索到最小元素的栈。

push(x) -- 将元素 x 推入栈中。
pop() -- 删除栈顶的元素。
top() -- 获取栈顶元素。
getMin() -- 检索栈中的最小元素
*/
#include <stdio.h>
#include <stdlib.h>
#define SIZE 20
#define INFINITY 2147483647
typedef struct
{
    // 主栈
    int main_stack[SIZE];
    // 最小栈
    int deputy_stack[SIZE];
    int top;
} MinStack;

/** initialize your data structure here. */

MinStack *minStackCreate()
{
    MinStack *minstack = (MinStack *)malloc(sizeof(MinStack));
    if (!minstack)
    {
        return NULL;
    }
    minstack->top = -1;
    for (int i = 0; i < SIZE; i++)
    {
        minstack->main_stack[i] = INFINITY;
        minstack->deputy_stack[i] = INFINITY;
    }
    return minstack;
}

void minStackPush(MinStack *obj, int x)
{
```

```
    int top = obj->top;
    obj->main_stack[top + 1] = x;
    // 比当前最小栈栈顶元素小或原本栈为空时直接入栈
    if (x < obj->deputy_stack[top] || top == -1)
    {
        obj->deputy_stack[top + 1] = x;
    }
    // 否则需要和最小栈栈顶元素进行比较，取较小者入最小栈
    else
    {
        int oldTop = obj->deputy_stack[top];
        obj->deputy_stack[top + 1] = oldTop;
    }
    obj->top = top + 1;
}

void minStackPop(MinStack *obj)
{
    if (obj->top >= 0)
    {
        obj->top--;
    }
}

int minStackTop(MinStack *obj)
{
    if (obj->top >= 0)
    {
        return obj->main_stack[obj->top];
    }
    else
    {
        return INFINITY;
    }
}

int minStackGetMin(MinStack *obj)
{
    if (obj->top >= 0)
    {
        return obj->deputy_stack[obj->top];
    }
    else
    {
        return INFINITY;
    }
```

```c
}

void minStackFree(MinStack *obj)
{
    free(obj);
    obj = NULL;
}

/**
 * Your MinStack struct will be instantiated and called as such:
 * MinStack* obj = minStackCreate();
 * minStackPush(obj, x);

 * minStackPop(obj);

 * int param_3 = minStackTop(obj);

 * int param_4 = minStackGetMin(obj);

 * minStackFree(obj);
*/

int main()
{
    MinStack *minstack = minStackCreate();
    minStackPush(minstack, -2);
    minStackPush(minstack, 0);
    minStackPush(minstack, -3);
    int param_4 = minStackGetMin(minstack);
    minStackPop(minstack);
    int param3 = minStackTop(minstack);
    int param_5 = minStackGetMin(minstack);
    minStackFree(minstack);
    printf("%d %d", param3, param_4);
    return 0;
}
```

矩阵置为0

```
// 给定一个 m x n 的矩阵，如果一个元素为 0，则将其所在行和列的所有元素都设为 0。请使用原地算法。
// 算法思想：
// 不能在每次遇到0时就将其所在行和所在列的所有元素置为0，这样会覆盖掉原本的值导致无法判断是原有的0还是
// 设置的0。比较好的做法是：设置两个变量分别记录第一行的元素和第一列的元素中是否出现过0，如果出现过则记为1
//  对于不在第一行和第一列的元素，如果遇到了0则将该行或者该列的第一个元素置为0；等所有的行都检查完毕后
// 1、根据第一行和第一列元素是否为0决定是否将其所在行和所在列置为0
// 2、根据这两个变量是否为0决定是否将第一行或者第一列置为0

// matrixSize表行数，matrixColSize表列数
void setZeroes(int **matrix, int matrixSize, int *matrixColSize)
{
    if (!matrix || matrixSize <= 0 || matrixColSize <= 0)
    {
        return;
    }
    // 第一行和第一列是否存在0元素
    int firstRowFlag = 0;
    int firstColFlag = 0;

    for (int i = 0; i < matrixSize; i++)
    {
        for (int j = 0; j < *matrixColSize; j++)
        {
            // 第一行和第一列之外的元素存在0元素
            if (i != 0 && j != 0 && matrix[i][j] == 0)
            {
                matrix[0][j] = 0;
                matrix[i][0] = 0;
            }
            // 第一行或第一列出现0元素
            else if (matrix[i][j] == 0)
            {
                firstRowFlag = i == 0 ? 1 : firstRowFlag;
                firstColFlag = j == 0 ? 1 : firstColFlag;
            }
        }
```

```
    }

    // 根据第一行和第一列元素是否为0将对应的整行或整列置为0
    for (int i = 1; i < matrixSize; i++)
    {
        for (int j = 1; j < *matrixColSize; j++)
        {
            if (matrix[i][0] == 0 || matrix[0][j] == 0)
            {
                matrix[i][j] = 0;
            }
        }
    }

    // 根据行列标记是否为1将第一行或第一列置为0
    if (firstRowFlag == 1)
    {
        for (int j = 0; j < *matrixColSize; j++)
        {
            matrix[0][j] = 0;
        }
    }

    if (firstColFlag == 1)
    {
        for (int j = 0; j < matrixSize; j++)
        {
            matrix[j][0] = 0;
        }
    }
}
```

岛屿数量

```c
// 给定一个由 '1'（陆地）和 '0'（水）组成的的二维网格，
// 计算岛屿的数量。一个岛被水包围，并且它是通过水平方向
// 或垂直方向上相邻的陆地连接而成的。你可以假设网格的四个边均被水包围。

// row和col是当前扩散的坐标，rows和cols是总的行数和列数
void infect(char **grid, int row, int col, int rows, int cols)
{
    if (row >= 0 && col >= 0 && row < rows && col < cols)
    {
        if (grid[row][col] == '1')
        {
            grid[row][col] = '2';
            // 上
            infect(grid, row - 1, col, rows, cols);
            // 下
            infect(grid, row + 1, col, rows, cols);
            // 左
            infect(grid, row, col - 1, rows, cols);
            // 右
            infect(grid, row, col + 1, rows, cols);
        }
    }
}

// 算法思想是:
// 从一个陆地开始(记为1的),将其标记为2表示已经访问,
// 递归从该陆地的上下左右四个点开始,如果是陆地则继续
// 以上操作直至所有的该区域陆地都被标记为已经访问过。
// 类似于广度优先搜索
// gridSize代表行数，gridColSize代表列数
int numIslands(char **grid, int gridSize, int *gridColSize)
{
    if (!grid)
    {
        return 0;
    }
    // 行
    int rows = gridSize;
    // 岛屿数量
    int count = 0;
```

```
    for (int i = 0; i < rows; i++)
    {
        for (int j = 0; j < *gridColSize; j++)
        {
            if (grid[i][j] == '1')
            {
                infect(grid, i, j, rows, *gridColSize);
                count++;
            }
        }
    }
    return count;
}
```

最大公约数

```c
// 最大公约数
#include <stdio.h>
int getMaxGcd(int big,int small){
    if(big % small == 0){
        return small;
    }
    // 记录除以2的次数
    int times = 0;
    while(big!=small){
        if(big % 2 == 0 && small % 2 == 0){
            big /= 1;
            small /= 2;
            times++;
        }else{
            int temp = big - small;
            if(temp >= small){
                big = temp;
            }else{
                big  = small;
                small = temp;
            }
        }
    }

    if(times == 0){
        return small;
    }else{
        return small * times * 2;
    }
}

int main(){
    int a,b;
    scanf("%d %d" ,&a,&b);
    // 始终使a是较大数，b是较小数
    if(a < b){
        int temp = a;
        a = b;
        b = temp;
    }
```

```
    printf("%d",getMaxGcd(a,b));
}
```

三数之和

```
#include <iostream>
#include <algorithm>
#include <vector>
using namespace std;
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        // 用于返回的数组
        vector<vector<int>> res;

        // 至少需要有 三个元素
        if(nums.size() <= 2)
            return res;

        // 对数组元素进行升序排序
        sort(nums.begin(),nums.end());

        // 算法思想是:先固定一个元素，然后从后面的元素中查找和等于第一个数相反数的
        // 两个数
        for(int i = 0;i < nums.size() - 2;i++){

            // 这里确保左边固定的第一个数没有重复，如果和之前的数重复了，那么必定会出现
重复的三元组
            if(i !=0 && nums[i] == nums[i - 1])
                continue;

            // 左右双指针，类似于求两个数的和，从左右两个方向查找
            int left = i + 1,right = nums.size() - 1;

            while(left < right){
                int sum = nums[i] + nums[left] + nums[right];

                // 当前三个数之和等于0,则加入返回数组中并将左右指针向中间靠拢
                if(sum == 0){
                    res.push_back({nums[i],nums[left],nums[right]});
                    left++;
                    right--;

                    // 这里左右指针变化后如果和变化之前的位置值相同，那么这三个数一定
是重复的
```

```
                    // 直接跳过直到和之前位置的值不同为止
                    while(left < right && nums[left - 1] == nums[left])
                        left++;
                    while(left < right && nums[right + 1] == nums[right])
                        right--;
                }else if(sum < 0){
                    left++;
                }else{
                    right--;
                }
            }

        }
        return res;

    }
};
```

旋转数组

```c
// 给定一个数组，将数组中的元素向右移动 k 个位置，其中 k 是非负数

// 方法一:时间换空间
void rotate_v1(int* nums, int numsSize, int k){
    // k = 0或 k是数组大小的倍数时旋转前后一样，直接返回
    if(!nums || numsSize <= 1 || k == 0 || (k % numsSize == 0)){
        return;
    }

    // 临时数组
    int *temp = (int*)malloc(sizeof(int) * numsSize);
    if(!temp){
        return;
    }
    for(int i = 0;i < numsSize;i++){
        temp[(i + k) % numsSize] = nums[i];
    }
    for(int j = 0;j < numsSize;j++){
        nums[j] = temp[j];
    }
    free(temp);
}

// 方法二: 每次讲数组向后移动一位，移动 k % n 次
// 时间复杂度为O(n * k),空间复杂度为O(1)
void rotate2(int* nums, int numsSize, int k){

    if(k % numsSize == 0)
        return;
    for(int i = 0;i < k % numsSize;i++){
        int temp = nums[numsSize - 1];
        for(int j = numsSize - 2;j >= 0;j--){
            nums[j+1] = nums[j];
        }
        nums[0] = temp;
    }

}

// 方法三:通过三次旋转实现,首先将整个数组转置，再将 0 ~ k - 1 和 k ~ n - 1 之间的
```

```c
// 部分转置
// 时间复杂度为O(n),空间复杂度为O(1)

void reverse(int* nums,int left,int right){
    while(left < right){
        nums[left] = nums[left] ^ nums[right];
        nums[right] = nums[right] ^ nums[left];
        nums[left] = nums[left] ^ nums[right];
        left++;
        right--;
    }
}
void rotate_v3(int* nums, int numsSize, int k){
    if(!nums || numsSize <= 1 || k == 0 || (k % numsSize == 0)){
        return;
    }
    k %= numsSize;
    reverse(nums,0,numsSize - 1);
    reverse(nums,0,k - 1);
    reverse(nums,k,numsSize - 1);
}
```

奇偶链表问题

```
// 给定一个单链表，把所有的奇数节点和偶数节点分别排在一起。请注意，这里的奇数节点和偶数

// 节点指的是节点编号的奇偶性，而不是节点的值的奇偶性。

// 请尝试使用原地算法完成。你的算法的空间复杂度应为 O(1)，时间复杂度应为 O(nodes)，nod
es 为节点总数。

struct ListNode {
      int val;
      struct ListNode *next;
};

struct ListNode* oddEvenList(struct ListNode* head){

    // 链表为空或只有一个节点或只有两个节点的情况
    if(!head || !head->next || !head->next->next){
        return head;
    }

    // 三个以上节点的情况
    // 奇偶链表的起始节点
    struct ListNode *oddHead = head,*evenHead = oddHead->next;
    // 奇偶链表的当前位置
    struct ListNode *oddPos = oddHead,*evenPos = evenHead;

    // 只有奇数个节点时最后 evenPos = NULL;只有偶数个节点时最后evenPos->next == N
ULL;
    while(evenPos && evenPos->next){
        oddPos->next = evenPos->next;
        oddPos = oddPos->next;
        evenPos->next = oddPos->next;
        evenPos = evenPos->next;
    }
    // 将偶数链表连接到奇数链表的后面
    oddPos->next = evenHead;
    return oddHead;
}
```

具有最大和的连续子序列

给定一个整数数组 nums ， 找到一个具有最大和的连续子数组(子数组最少包含一个元素),返回其最大和。

```
int maxSubArray(int* nums, int numsSize){
    if(!nums || numsSize <= 0){
        return 0;
    }
    // 动态规划辅助数组，dp[i]表示以第i位置字符结束的
    // 最大子序和
    int *dp = (int*)malloc(sizeof(int) * numsSize);
    if(!dp){
        return 0;
    }
    dp[0] = nums[0];

    // 如果当前元素值加上前面元素的最大和比自身值大，则其和作为新的最大和，否则当前元素
    // 单独作为一个子序列
    for(int i = 1;i < numsSize;i++){
         dp[i] = nums[i] < dp[i - 1] + nums[i] ? dp[i - 1] + nums[i] : nums
[i];
    }

    // 寻找最大和
    int maxSum = dp[0];
    for(int j = 1;j < numsSize;j++){
        if(dp[j] > maxSum){
            maxSum = dp[j];
        }
    }
    return maxSum;
}
```

颜色分类

> 给定一个包含红色、白色和蓝色，一共 n 个元素的数组，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。此题中，我们使用整数 0、 1 和 2 分别表示红色、白色和蓝色。

颜色分类

> 计数排序法

```
void sortColors_v1(int* nums, int numsSize){
    int temp[3] = {0,0,0};
    for(int i = 0;i < numsSize;i++){
        temp[nums[i]]++;
    }
    int index = 0;
    for(int j = 0;j < 3;j++){
        while(temp[j]!=0){
            nums[index++] = j;
            temp[j]--;
        }
    }
}
```

> 双指针法

```
void sortColors_v2(int* nums, int numsSize){
    // 从最左边和最右边开始扫描
    int left = 0,right = numsSize - 1;

    // 跳过左侧连续的0
    while(left < right && nums[left] == 0){
        left++;
    }
    // 跳过右侧连续的2
    while(left < right && nums[right] == 2){
        right--;
    }

    // 指针left始终指向左侧第一个非0，right始终指向右侧第一个非2
    for(int i = left;i <= right;i++){
```

```
            // 遇到0将其与left指向的非0元素交换,此时从left交换过来的一定是
            // 1,所以直接从下一个i比较即可
            if(nums[i] == 0){
                int temp = nums[i];
                nums[i] = nums[left];
                nums[left] = temp;
                left++;
            }
            // 遇到2将其与right指向的非2元素交换，此时right交换过来的
            // 可能是1或0,所以i需要回退再比较一次
            else if(nums[i] == 2){
                int temp = nums[i];
                nums[i] = nums[right];
                nums[right] = temp;
                right--;
                i--;
            }
        }
    }
```

反序数

```c
// 反序数
#include<stdio.h>

int main(){

    for(int i = 1001 ;i < 10000;i ++){
        int temp = i * 9;
        int first = temp % 10;
        temp = temp / 10;
        int second = temp % 10;
        temp = temp / 10;
        int third = temp % 10;
        temp = temp / 10;
        int fourth = temp % 10;

        if(temp / 10  !=0 ){
            continue;
        }

        if(first * 1000 + second * 100 + third * 10 + fourth == i){
            printf("%d\n",i);
        }

    }
    return 0;
}
```

在排序数组中查找元素的第一个和最后一个位置

在排序数组中查找元素的第一个和最后一个位置

给定一个按照升序排列的整数数组 nums，和一个目标值 target。找出给定目标值在数组中的开始位置和结束位置。

你的算法时间复杂度必须是 O(log n) 级别。

如果数组中不存在目标值，返回 [-1, -1]。

```
int* searchRange(int* nums, int numsSize, int target, int* returnSize){
    int left = 0,right = numsSize - 1;

    // 返回数组，保存第一次出现和最后一次出现的索引
    int *ret = (int*)malloc(sizeof(int) * 2);
    ret[0] = -1;
    ret[1] = -1;
    while(left <= right){
        int mid = (left + right) / 2;
        // 目标数在mid左边
        if(nums[mid] > target){
            right = mid - 1;
        }
        // 目标数在mid右边
        else if(nums[mid] < target){
            left = mid + 1;
        }

        // 当前数等于目标数，还需要继续向左边查找第一个，向右边查找最后一个
        else{
            // 左边第一个
            while(nums[mid] == target && mid >= left){
                mid--;
            };
            ret[0] = mid + 1;
            int mid = (left + right) / 2;
            // 右边最后一个
```

```
            while(nums[mid] == target && mid <= right){
                mid++;
            }
            ret[1] = mid - 1;
            break;
        }

    }
    return ret;
}
```

无重复字符的最长子串

无重复字符的最长子串

```
#include <string.h>

int lengthOfLongestSubstring(char * s){
    // 辅助数组,记录每个字符最近一次出现的位置
    int map[128] = {-1};

    memset(map,-1,sizeof(map));

    // 最大长度的子串
    int longestLength = 0;

    if(!s){
        return 0;
    }
    // 串的长度
    int size = 0;
    for(int i = 0;s[i]!='\0';i++){
        size++;
    }

    // left、right分别表示滑动窗口的两边
    for(int left = 0,right = 0;right < size;right++){
        // 如果当前字符在滑动窗口中，则需要将上次出现的字符排除到窗口外
        if(map[s[right]] >= left){
            left = map[s[right]] + 1;
        }

        // 更新最近出现的位置
        map[s[right]] = right;

        // 更新最大长度
        longestLength =  longestLength > right - left + 1 ? longestLength :
 right - left + 1;
    }
    return longestLength;
}
```

模式匹配

```
// 普通的模式匹配
int plainMatch(char *str,char *pattern){
    if(!str || !pattern){
        return -1;
    }

    int i,j;
    for(i = 0;str[i]!='\0';){
        for(j = 0;pattern[j] != '\0';){
            if(str[i] == pattern[j]){
                i++;
                j++;
            }else{
                i = i - j + 1;
                j = 0;
                break;
            }
        }
        if(pattern[j] == '\0'){
            break;
        }
    }
    if(pattern[j] == '\0'){
        return i - j;
    }else
    {
        return -1;
    }
}
```

去除排序数组中的重复元素

```c
int removeDuplicates(int* nums, int numsSize){
    if(!nums || numsSize <=0){
        return numsSize;
    }

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

        // 寻找第一个不相同的元素
        for(j = i + 1;j < numsSize && nums[j] == nums[i];j++){

        }
        if(j - i  == 1){
            continue;
        }

        // 将后面的元素向前移动
        for(int k = j;k < numsSize;k++){
            nums[k - j + i + 1] = nums[k];
        }

        // 修改数组长度
        numsSize-= (j - i - 1);
    }

    return numsSize;
}
```

最长回文子串

```c
// 给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。

// 双向扩散方法
// length是字符串长度,left和right分别是扩散的起点,maxStart和maxEnd是回文串的起始索引
void diffusion(char *s,int length,int left,int right,int *maxStart,int *maxEnd){
    if(!s){
        return;
    }
    // 当前两个位置字符相等继续向左右扩散
    while(left >= 0 && right <= length - 1 && s[left] == s[right]){
        left--;
        right++;
    }

    // 如果比最大的长度还长，则更新为新的最长汇文子串
    if(right - left - 1 > *maxEnd - *maxStart + 1){
        *maxStart = left + 1;
        *maxEnd = right - 1;
    }

}
char * longestPalindrome(char * s){

    // 字符串的长度
    int length = 0;
    for(int i = 0;s[i]!='\0';i++){
        length++;
    }

    // 空串或长度为1是回文串
    if(length <= 1){
        return s;
    }

    // 最大回文子串和起始索引
    int maxStart = 0;
    int maxEnd = 0;
```

```
    for(int i = 0;i < length;i++){
        // aba形式
        diffusion(s,length,i,i,&maxStart,&maxEnd);
        // abba形式
        diffusion(s,length,i,i + 1,&maxStart,&maxEnd);
    }

    // 复制到返回数组
    char *ret = (char*)malloc(sizeof(char) * (maxEnd - maxStart + 2));
    for(int i = 0,j = maxStart;j <= maxEnd;j++,i++){
        ret[i] = s[j];
    }
    ret[maxEnd - maxStart + 1] = '\0';
    return ret;
}
```

指定路径和的所有路径

```
#define  SIZE 20

typedef struct TreeNode {
  int val;
  struct TreeNode *left;
  struct TreeNode *right;
}TreeNode;

// 打印指定路径和的所有路径
void getPathOfSum(TreeNode *root,int sum){
    if(root == NULL){
        return;
    }

    // 辅助栈
    int top = -1;
    TreeNode* stack[SIZE];

    TreeNode *pos = root;

    // 最近访问的节点
    TreeNode *lastVisited = NULL;

    // 临时记录路径和
    int temp = 0;

    while(pos || top!=-1){

        // 当前节点不空，入栈，更新路径和
        if(pos){
            temp += pos->val;
            stack[++top] = pos;

            // 如果是叶节点且路径和等于指定值则可以打印一条路径
            if(!pos->left && !pos->right && temp == sum){
                for(int i = 0;i <= top;i++){
                    printf("%d ",stack[i]->val);
                }
                printf("\n");
            }
```

```
            // 向左访问
            pos = pos->left;

        }else{
            pos = stack[top];
            // 右子树存在且未被访问则访问右节点
            if(pos->right != NULL && pos->right!=lastVisited){
                pos = pos->right;
            }
            // 右子树不存在或已经访问过，访问当前节点并出栈，将路径和还原
            else{
                top--;
                temp -= pos->val;
                lastVisited = pos;
                pos = NULL;
            }
        }
    }
}
```

打印元素的祖先节点

```
typedef struct TreeNode {
  int val;
  struct TreeNode *left;
  struct TreeNode *right;
}TreeNode;
void getAncestor(TreeNode *root, int x){
      // 栈
      int top = -1;
      TreeNode* stack[SIZE];

      TreeNode *pos = root;

      // 最近访问的节点
      TreeNode *lastVisited = NULL;

      while(pos || top!=-1){
            // 当前节点不空且不是指定节点，继续向左寻找
            if(pos && pos->val!=x){
                  stack[++top] = pos;
                  pos = pos->left;
            }
            // 当前节点时指定节点，则栈里的元素都是其祖先节点
            else if(pos){
                  if(top == -1){
                        printf("当前节点是根节点,没有祖先\n");
                  }
                  while(top > -1){
                        printf("%d ",stack[top--]->val);
                  }
                  return;
            }
            // 当前节点为空
            else if(!pos){
                  pos = stack[top];
                  // 右子树存在且未被访问
                  if(pos->right && lastVisited != pos->right){
                        pos = pos->right;
                  }
                  // 右子树不存在或已经访问过，访问当前节点并出栈
                  else{
```

```
                        pos = stack[top--];
                        lastVisited = pos;
                        pos = NULL;
                    }
                }
            }
            printf("节点不存在\n");
}
```

删除倒数第K个节点

```c
typedef struct ListNode {
      int val;
      struct ListNode *next;
}ListNode;

ListNode* removeNthFromEnd(struct ListNode* head, int n){
    ListNode *fast = head,*slow = head;
    int i;
    for(i = 0;i < n && fast;i++){
        fast = fast->next;
    }

    // 节点数量不足n
    if(i < n){
        return false;
    }
    // 指向待删除节点
    ListNode *temp;
    // 删除的是头结点
    if(!fast){
        temp = head;
        head = head->next;
        free(temp);
        return head;
    }

    // fast指针指向最后一个节点时slow指向待删除节点的前驱节点
    while(fast->next){
        fast = fast->next;
        slow = slow->next;
    }

    temp = slow->next;
    slow->next = slow->next->next;
    free(temp);
    return head;
}
```

判断回文链表

```
typedef struct ListNode {
      int val;
      struct ListNode *next;
}ListNode;
// 算法思想:
// 1. 采用快慢指针方式找到链表的中间节点
// 2. 将后面的半段链表采用迭代的方式逆置
// 3. 从两端链表头部进行比较，如果完全符合则是回文链表
bool isPalindrome(struct ListNode* head){

    // 链表为空或只有一个节点
    if(!head || !head->next){
        return true;
    }

    // 只有两个节点
    if(!head->next->next){
        if(head->val == head->next->val){
            return true;
        }else{
            return false;
        }
    }

    // 三个以上节点采用快慢指针
    ListNode *fast = head;
    ListNode *slow = head;

    // 后一条链表的开始
    ListNode *secondHead = NULL;
    while(fast->next){
        fast = fast->next->next;
        slow = slow->next;
        // 偶数个节点时快指针最终指向最后一个节点的后面，即空指针
        // 此时慢指针指向第二个链表的开始
        if(!fast){
            secondHead = slow;
            break;
        }
```

```
        // 奇数个节点时快指针最终指向最后一个节点
        if(fast && !fast->next){
            secondHead = slow->next;
            break;
        }
    }

    // 下面用迭代的方式转置后面一条链表,pos表示当前位置,
    ListNode *pos = secondHead;
    ListNode *pos_next = NULL;
    secondHead = NULL;
    while(pos){
        pos_next = pos->next;
        pos->next = secondHead;
        secondHead = pos;
        pos = pos_next;
    }

    // 逐个比较每个元素是否相等
    for(ListNode *l1 = head,*l2 = secondHead;l1 && l2;){
        if(l1->val == l2->val){
            l1 = l1->next;
            l2 = l2->next;
        }else{
            return false;
        }
    }

    return true;
}
```

自底向上、从右向左遍历二叉树

```
#define SIZE 20

typedef struct TreeNode {
  int val;
  struct TreeNode *left;
  struct TreeNode *right;
}TreeNode;

// 主要思想是:采用层次遍历的方式，出队时入栈，访问完毕后栈里的顺序是从上到下，从左往右的
// 逆向输出即可
void down2up_right2left(TreeNode *root){
    if(!root){
        return;
    }
    // 辅助栈
    TreeNode* stack[SIZE];
    int top = -1;
    // 辅助队列
    int head = -1,tail = -1;
    TreeNode* queue[SIZE];

    queue[++head] = root;
    while(head!=tail){
        TreeNode *temp = queue[++tail];
        stack[++top] = temp;
        if(temp->left){
            queue[++head] = temp->left;
        }
        if(temp->right){
            queue[++head] = temp->right;
        }
    }

    while(top!=-1){
        printf("%d\t",stack[top--]->val);
    }
}```
```

完数和盈数

一个数如果恰好等于它的各因子(该数本身除外)子和，如：6=3+2+1。则称其为“完数”；若因子之和大于该数，则称其为“盈数”。求出2到60之间所有“完数”和“盈数”。

```
// 完数和盈数

#include<stdio.h>

int getSum(int num){
    int sum = 0;
    for(int i = 1;i <=  num / 2;i ++){
        if(num % i == 0){
            sum += i;
        }
    }
    return sum;
}

int main(){

    int arr[61] = {0};

    for(int i  = 2;i <=  60;i ++){
        //  完数
        if(getSum(i) == i){
            arr[i] = 1;
        }
        // 盈数
        else if(getSum(i) > i){
            arr[i] = 2;
        }
    }

    printf("E:");

    for(int i  = 2;i <=  60;i ++){
        //  完数
        if(arr[i] == 1){
            printf(" %d",i);
        }
    }
```

```
    printf("\nG:");

    for(int i  = 2;i <=  60;i ++){
        //  盈数
        if(arr[i] == 2){
            printf(" %d",i);
        }
    }

    return 0;

}
```

二叉树遍历

数据类型定义

```
#define SIZE 20

typedef struct TreeNode {
  int val;
  struct TreeNode *left;
  struct TreeNode *right;
}TreeNode;
```

递归方式

```
// 前序递归遍历
void preTraverseRecursion(TreeNode *root){
    if(!root){
        return;
    }
    printf("%d",root->val);
    preTraverseRecursion(root->left);
    preTraverseRecursion(root->right);
}

// 中序递归遍历
void inTraverseRecursion(TreeNode *root){
    if(!root){
        return;
    }
    inTraverseRecursion(root->left);
    printf("%d",root->val);
    inTraverseRecursion(root->right);
}

// 后序递归遍历
void postTraverseRecursion(TreeNode *root){
    if(!root){
        return;
```

```
    }
    postTraverseRecursion(root->left);
    postTraverseRecursion(root->right);
    printf("%d",root->val);
}
```

非递归方式

```
// 前序非递归遍历
void preTraverseNonRecursion(TreeNode *root){

    if(!root){
        return;
    }
    // 辅助栈
    int top = -1;
    TreeNode* stack[SIZE];

    stack[++top] = root;

    while(top!=-1){
        TreeNode *temp = stack[top--];
        printf("%d ",temp->val);
        if(temp->right){
            stack[++top] = temp->right;
        }
        if(temp->left){
            stack[++top] = temp->left;
        }
    }
}

// 中序非递归遍历
void inTraverseNonRecursion(TreeNode *root){
    if(!root){
        return;
    }
    // 辅助栈
    int top = -1;
    TreeNode* stack[SIZE];

    // 当前节点
    TreeNode *pos = root;
```

```
    while(pos || top!=-1){
        if(pos){
            stack[++top] = pos;
            pos = pos->left;
        }else{
            pos = stack[top--];
            printf("%d",pos->val);
            pos = pos->right;
        }
    }
}

// 后序非递归遍历
void postTraverseNonRecursion(TreeNode *root){
    if(!root){
        return;
    }
    // 辅助栈
    int top = -1;
    TreeNode* stack[SIZE];
    // 当前访问的节点
    TreeNode *pos = root;
    // 记录上一步访问的节点,据此确定下一步是访问根节点还是访问右子树
    TreeNode *lastVisited = NULL;
    while(pos || top!=-1){
        // 当前节点不空则一直访问到最左边
        if(pos){
            stack[++top] = pos;
            pos = pos->left;
        }else{
            pos = stack[top];
            // 当前节点有右孩子且未被访问，则转到右子树
            if(pos->right && pos->right != lastVisited){
                pos = pos->right;
                stack[++top] = pos;
                pos = pos->left;
            }
            // 没有右子树或右子树已访问，则访问当前节点
            else{
                pos = stack[top--];
                lastVisited = pos;
                printf("%d ",pos->val);
                pos = NULL;
            }
        }
    }
```

```
}
```

反转链表

```
typedef struct ListNode {
      int val;
      struct ListNode *next;
}ListNode;

// 迭代方式
ListNode* reverseListRecursion(struct ListNode* head){

    // 链表为空或只有一个节点
    if(!head || !head->next){
        return head;
    }
    // 返回链表的头指针
    ListNode *retHead = NULL;

    // 当前位置和下一位置
    ListNode *pos = head;
    ListNode *next_pos = head->next;

    while(pos){
        next_pos = pos->next;
        pos->next = retHead;
        retHead = pos;
        pos = next_pos;
    }
    return retHead;
}

// 递归方式
ListNode* reverseListIternation(struct ListNode* head){
    // 链表为空或只有一个节点
    if(!head || !head->next){
        return head;
    }else{
        // 转置链表头部
        ListNode *retHead = reverseListIternation(head->next);
        // 转置链表尾部
        ListNode *retEnd = retHead;
        while(retEnd->next){
            retEnd = retEnd->next;
```

```
        }
        retEnd->next = head;
        head->next = NULL;
        return retHead;

    }
}
```

数组的第K小元素

```
// 递归方式
int getKthMinimumRecursion(int arr[],int low,int high,int k){

    // 边界检查
    if(k < 1 || k > high + 1)
        return -2147483648;

    int lowTemp =  low,highTemp = high;

    srand((unsigned)time(NULL));

    // 随机选择枢纽位置
    int pivotIndex = low + rand() % (high - low + 1);
    int temp = arr[pivotIndex];
    arr[pivotIndex] = arr[low];
    arr[low] = temp;

    // 确定枢纽值的位置
    while(low < high){
        while(low < high && arr[high] >= temp){
            high--;
        }
        arr[low] = arr[high];
        while(low < high && arr[low] <= temp){
            low++;
        }
        arr[high] = arr[low];
    }

    arr[low] = temp;

    // 枢纽值是第k个元素
    if(low + 1 == k){
        return arr[low];
    }
    // 第k元素在枢纽值之前
    else if(low  + 1 > k){
        return getKthMinimumRecursion(arr,lowTemp,low - 1,k);
    }else{
        return getKthMinimumRecursion(arr,low + 1,highTemp,k);
```

```
    }
}
// 随机生成随机索引并返回中枢元素的位置
int getPivot(int arr[],int left,int right){
    srand((unsigned)time(NULL));
    int pivotIndex = rand() % (right - left + 1) + left;
    int temp = arr[pivotIndex];
    arr[pivotIndex] = arr[left];
    arr[left] = temp;
    while(left < right){
        while(left < right && arr[right] >= temp){
            right--;
        }
        arr[left] = arr[right];
        while(left < right && arr[left] <= temp){
            left++;
        }
        arr[right] = arr[left];
    }
    arr[left] = temp;
    return pivotIndex;
}

// 迭代方式
int getKthMinimumIteration(int arr[],int size,int k){
    // 边界检查
    if(k < 1 || k > size){
        return -2147483648;
    }
    int left = 0;
    int right = size - 1;
    int pivot;
    while(left <= right){
        pivot = getPivot(arr,left,right);
        if(pivot + 1 == k){
            return pivot;
        }else if(pivot + 1 < k){
            left = pivot + 1;
        }else{
            right = pivot - 1;
        }
    }
    return pivot;
}
```

移动 0

```c
// 给定一个数组 nums，编写一个函数将所有 0 移动到数组的末尾，同时保持非零元素的相对顺序
。
void moveZeroes_v1(int* nums, int numsSize){
    // 记录非0元素的位置
    int k = 0;
    for(int i = 0;i < numsSize;i++){
        if(nums[i]!=0){
            nums[k++] = nums[i];
        }
    }

    while(k < numsSize){
        nums[k++] = 0;
    }
}

void moveZeroes_v2(int* nums, int numsSize){
    // 记录0元素的个数
    int k = 0;
    for(int i = 0;i < numsSize;i++){
        if(nums[i] == 0){
            k++;
        }else{
            nums[i - k] = nums[i];
        }
    }

    for(int i = numsSize - k + 1;i < numsSize;i++){
        nums[i] = 0;
    }
}
```

合并有序链表

```
// 将两个有序链表合并为一个新的有序链表并返回。新链表是通过拼接
// 给定的两个链表的所有节点组成的。
typedef struct ListNode {
    int val;
    struct ListNode *next;
 }ListNode;

 struct ListNode* mergeTwoLists( ListNode* l1, ListNode* l2){
     if(!l1){
         return l2;
     }
     if(!l2){
         return l1;
     }

     // 分别指向待合并的两个链表的当前位置
     ListNode *pos_l1 = l1,*pos_l2 = l2;

     // 返回链表的头结点
     ListNode *retHead = (ListNode*)malloc(sizeof(ListNode));
     if(retHead == NULL){
         return NULL;
     }
     // 合并链表的当前位置
     ListNode *pos_ret = retHead;

     // 将两个链表较小的节点拆下来挂到新链表上
     while(pos_l1 && pos_l2){
         if(pos_l1->val < pos_l2->val){
             pos_ret->next = pos_l1;
             pos_ret = pos_l1;
             pos_l1 = pos_l1->next;
         }else{
            pos_ret->next = pos_l2;
             pos_ret = pos_l2;
             pos_l2 = pos_l2->next;
         }
     }

     // 处理剩余的节点
```

```
    if(pos_l1){
        pos_ret->next = pos_l1;
    }
    if(pos_l2){
        pos_ret->next = pos_l2;
    }

    ListNode *ret = retHead->next;
    free(retHead);
    return ret;
}
```

反转字符串

```
/**
  编写一个函数，其作用是将输入的字符串反转过来。输入字符串以字符数组 char[] 的形式给出。

不要给另外的数组分配额外的空间，你必须原地修改输入数组、使用 O(1) 的额外空间解决这一问题
。

你可以假设数组中的所有字符都是 ASCII 码表中的可打印字符。

*/

#include<stdio.h>



void reverseString(char* s, int sSize){
    for(int i = 0,j = sSize - 1;i < j;i++,j--){
        s[i] = s[i] ^ s[j];
        s[j] = s[j] ^ s[i];
        s[i] = s[j] ^ s[i];
    }
}
```

镜像对称的二叉树

```
// 树节点
typedef struct TreeNode{
    int data;
    struct TreeNode *left;
    struct TreeNode *right;
}TreeNode;

bool symmetric(TreeNode *left,TreeNode *right){

    // 两个节点都为空是对称的
    if(!left && !right){
        return true;
    }

    // 只有一个节点为空或两节点都不空但不相等是不对称的
    if(!left || !right || left->data != right->data){
        return false;
    }

    // 继续判断两个节点对称位置的节点
    return symmetric(left->left,right->right)
    && symmetric(left->right,right->left);
}

bool isSymmetricTree(TreeNode *root){
    if(root == NULL){
        return true;
    }
    return symmetric(root->left,root->right);
}
```

二叉树的最大深度

```c
#define SIZE 30
// 树节点
typedef struct TreeNode{
    int data;
    struct TreeNode *left;
    struct TreeNode *right;
}TreeNode;

// 递归方式
int getMaxDepthRecursion(TreeNode *root){
    if(!root){
        return 0;
    }

    int leftHeight = getMaxDepthRecursion(root->left);
    int rightHeight = getMaxDepthRecursion(root->right);
    return (leftHeight < rightHeight ? rightHeight : leftHeight) + 1;
}

// 层序遍历方式
int getMaxDepthLevel(TreeNode *root){
    int depth = 0;
    if(root == NULL){
        return depth;
    }

    // 辅助队列
    int head = -1,tail = -1;
    TreeNode* queue[SIZE];

    queue[++head] = root;

    // 用于标记每一行的末尾和下一行的末尾
    TreeNode *thisEnd = root,*nextEnd = NULL;

    while(head != tail){
        tail = (tail+1) % SIZE;
        TreeNode *temp = queue[tail];
        if(temp->left){
            head = (head + 1) % SIZE;
```

```
            queue[head] = temp->left;
            nextEnd = temp->left;
        }
        if(temp->right){
            head = (head + 1) % SIZE;
            queue[head] = temp->right;
            nextEnd = temp->right;
        }
        if(temp == thisEnd){
            thisEnd = nextEnd;
            // 每行结束时深度加一
            depth++;
        }
    }
    return depth;
}
```

二叉树的最大宽度

```c
#include<stdio.h>
#include<stdlib.h>
// 队列大小
#define SIZE 30
// 树节点
typedef struct TreeNode{
    int data;
    struct TreeNode *left;
    struct TreeNode *right;
}TreeNode;

// 计数方式
int getMaxWidth_1(TreeNode *root){

    if(root == NULL){
        return 0;
    }

    // 辅助循环队列
    TreeNode* queue[30];
    int head = -1,tail = -1;

    queue[++head] = root;

    // maxwidth记录最大宽度，count记录每一层的宽度
    int maxWidth = 1,count = 1;

    while(head != tail){
        tail = (tail + 1) % SIZE;
        TreeNode *temp = queue[tail];
        count--;
        if(temp->left){
            head = (head + 1) % SIZE;
            queue[head] = temp->left;
        }
        if(temp->right){
            head = (head + 1) % SIZE;
            queue[head] = temp->right;
        }
        if(count == 0){
```

```
            count = (head - tail) % SIZE;
            maxWidth = maxWidth > count ? maxWidth : count;
        }
    }
    return maxWidth;
}

// 指针方式
int getMaxWidth_2(TreeNode *root){

    if(root == NULL){
        return 0;
    }

    // 辅助循环队列
    TreeNode* queue[30];
    int head = -1,tail = -1;

    queue[++head] = root;

    // maxwidth记录最大宽度，count记录每一层的宽度
    int maxWidth = 1,count = 0;

    // 分别指向当前行的最后一个元素和下一行的最后一个元素
    TreeNode *rowEnd = root,*nextRowEnd = NULL;

    while(head != tail){
        tail = (tail + 1) % SIZE;
        TreeNode *temp = queue[tail];
        if(temp->left){
            head = (head + 1) % SIZE;
            queue[head] = temp->left;

            // nextRowEnd 指向当前行的最后一个元素
            nextRowEnd = temp->left;
            count++;
        }
        if(temp->right){
            head = (head + 1) % SIZE;
            queue[head] = temp->right;

            //nextRowEnd 指向当前行的最后一个元素
            nextRowEnd = temp->right;
            count++;
        }
        if(temp == rowEnd){
```

```
            maxWidth = maxWidth > count ? maxWidth : count;
            rowEnd = nextRowEnd;
            count = 0;
        }
    }
    return maxWidth;
}
```

二叉树层序遍历

```c
#include<stdio.h>
#include<stdlib.h>
// 队列大小
#define SIZE 30
// 树节点
typedef struct TreeNode{
    int data;
    struct TreeNode *left;
    struct TreeNode *right;
}TreeNode;

int* levelTraverse(TreeNode *root,int returnSize){
    if(root == NULL){
        return NULL;
    }

    // 返回节点数据数组
    int* ret = (int*)malloc(sizeof(int) * returnSize);
    if(ret == NULL){
        return NULL;
    }

    // 辅助循环队列
    TreeNode* queue[30];
    int top = -1,tail = -1;

    queue[++top] = root;

    int pos = 0;

    while(top != tail){
        tail = (tail + 1) % SIZE;
        TreeNode *temp = queue[tail];
        ret[pos++] = temp->data;
        if(temp->left){
            top = (top + 1) % SIZE;
            queue[top] = temp->left;
        }
        if(temp->right){
            top = (top + 1) % SIZE;
```

```
            queue[top] = temp->right;
        }
    }

    return ret;
}
```

合并有序数组

```
// 给定两个有序整数数组 nums1 和 nums2，将 nums2 合并到 nums1 中，使得 num1 成为一个有序数组。

// 说明:

// 初始化 nums1 和 nums2 的元素数量分别为 m 和 n。
// 你可以假设 nums1 有足够的空间（空间大小大于或等于 m + n）来保存 nums2 中的元素。


#include <stdio.h>
void merge(int* nums1, int nums1Size, int m, int* nums2, int nums2Size, int
 n){

    // i指向第一个数组,j指向第二个数组,k用于记录合并后的位置
    int i = m - 1,j = n - 1,k = m + n - 1;
    for(;i >= 0 && j >= 0;){
        if(nums1[i] > nums2[j]){

            nums1[k--] = nums1[i--];
        }else{

            nums1[k--] = nums2[j--];
        }
    }

    while(i >= 0){

        nums1[k--] = nums1[i--];

    }

    while(j >= 0){

        nums1[k--] = nums2[j--];

    }
}
```

双指针判断回文串

```
// 给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写。

// 说明：本题中，我们将空字符串定义为有效的回文串。

#include<stdio.h>
#include<stdbool.h>
#include<ctype.h>


// 判断两个字符是否是相同的数字或者大小写无关的字母
bool isSameCharOrDigitIgnoreCase(char ch1,char ch2){
    // 相等的数字
    if(isdigit(ch1) && isdigit(ch2) && ch1 == ch2){
        return true;
    }

    // 大小写无关的字母
    if(isalpha(ch1) && isalpha(ch2)  && (ch1 == ch2 || ch1 - 'A' == ch2 - '
a'  || ch1 -  'a' ==  ch2 - 'A')){
        return true;
    }
    return false;
}



bool isPalindrome(char * s){
    int i;
    for(i = 0;s[i]!='\0';i++){

    }

    for(int left = 0,right = i - 1;left < right;left++,right--){

        // 跳过遇到的不合法字符

        while(!isalnum(s[left]) && left < right){
            left++;
        }
```

```
        while(!isalnum(s[right]) &&  left < right){
            right--;
        }

        // 只要有相应位置的两个字符不符合要求即退出
        if(left < right && !isSameCharOrDigitIgnoreCase(s[left],s[right])){
            return false;
        }
    }
    return true;
}

int main(){
    printf("%d\n",isPalindrome("A man, a plan, a canal: Panama") );
    return 0;
}
```

fizzbuzz

写一个程序，输出从 1 到 *n* 数字的字符串表示。

1. 如果 *n* 是3的倍数，输出“Fizz”；

2. 如果 *n* 是5的倍数，输出“Buzz”；

3.如果 *n* 同时是3和5的倍数，输出 “FizzBuzz”。

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

char * transferToString(int num){
    char arr[11];
    int length = 0;
    int temp = num;
    while(temp!=0){
        arr[length] = (temp % 10) - 0 + '0';
        temp /= 10;
        length++;
    }

    char *ret = (char *)malloc(sizeof(char) * (length + 1));
    for(int j = 0,k = length - 1;j < length;j++,k--){
        ret[j] = arr[k];
    }
    ret[length] = '\0';
    return ret;
 }

char ** fizzBuzz(int n, int* returnSize){
    char **ret = (char**)malloc(sizeof(char*) * n);
    for(int i = 1;i <= n;i++){
        if(i % 3 == 0 && i % 5 == 0){
            ret[i - 1] = "FizzBuzz";
        }else if(i % 3 == 0){
            ret[i - 1] = "Fizz";
        }else if(i % 5 == 0){
            ret[i - 1] = "Buzz";
        }else{
```

```
            ret[i - 1] = transferToString(i);
        }
    }

    *returnSize = n;
    return ret;
}
```

第一个不重复的字符

```c
// 给定一个字符串，找到它的第一个不重复的字符，并返回它的索引。如果不存在，则返回 -1。
// 注意事项：您可以假定该字符串只包含小写字母。
#include<stdio.h>



int firstUniqChar(char * s){

    // 键值对的映射，key为字母的ASCII码减去一个偏移量后的值，该偏移量恰好使得小写字母a映射到
    // 索引0.value为出现的次处，默认是0表示没有出现过。
    int map['z' - 'a' + 1] = {0};

    // 每次遍历到一个字符将其次数加1
    for(int i = 0;s[i] != '\0';i++){
        map[s[i] - 'a']++;
    }

    // 最后返回的索引，如果不存在则返回-1
    int index = -1;

    // 再次遍历字符串，如果遇到一个只出现了一次的字符则跳出并返回
    // s[k] - 'a' 是该字符在映射中的key
    for(int k = 0; s[k] != '\0';k++){
        if(map[s[k] -  'a'] == 1){
            index = k;
            break;
        }
    }
    return index;
}
```

爬楼梯

简单的动态规划思想

```
// 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?

// 注意：给定 n 是一个正整数。

#include <stdio.h>
#include <stdlib.h>

int climbStairs(int n){
    if(n ==  0)
        return 0;
    if(n == 1)
        return 1;

    int *temp  =  (int*)malloc(sizeof(int) * (n+1));
    temp[0] = 1;
    temp[1] = 1;

    for(int i = 2;i <= n;i++){
        temp[i] =  temp[i - 1] + temp[i - 2];
    }

    return temp[n];

}
```

将奇数移动到偶数前面

```
#include <stdio.h>
#include <stdlib.h>
// 将奇数移动到偶数前面,采用快排的思路两个指针同时从两端移动
void exchangeOvenOddPosition(int arr[],int count){
    if(!arr || count <= 0)
        return;
    int left = 0,right = count - 1,temp;
    while(left < right){
        while(left < right && arr[right] % 2 == 0)
            right--;

        while(left < right && arr[left] % 2 == 1)
            left++;

        if(left < right){
            int temp = arr[left];
            arr[left] = arr[right];
            arr[right] = temp;
            left++;
            right--;
        }
    }
}
```

双向冒泡排序

```
// 交换
void swap(int *a,int *b){
    int temp = *a;
    *a = *b;
    *b = temp;
}
// 升序
void doubleDirectionBubbleSortUp(int arr[],int size){
    if(!arr || size <= 0)
        return;

    // left、right分别指向第一个位置和最后一个位置
    int left = 0,right = size - 1;

    // 是否交换过
    int flag = 0;

    // 每一轮交换能确定开头和结尾两个位置的元素，后面每次只需要对中间的几个数进行
    // 排序即可
    while(left < right){

        // 最大元素到末尾
        for(int i = left;i < right;i++){
            if(arr[i] > arr[i+1]){
                flag = 1;
                swap(&arr[i],&arr[i+1]);
            }
        }
        if(flag == 0)
            break;
        right--;

        // 最小元素到开头
        for(int j = right;j > 0;j--){
            if(arr[j] < arr[j - 1]){
                swap(&arr[j],&arr[j - 1]);
            }
        }
        left++;
    }
```

```
}
```

优先级队列

```c
// 优先级队列，按优先级从高向低
#include <stdio.h>
#include <stdlib.h>
#define DATATYPE int

// 优先级队列数据元素
typedef struct element{
    DATATYPE data;
    int priority;
}element;

// 优先级队列
typedef struct PriorityQueue{
    element *unit;

    // 实际元素个数
    int size;
    // 总容量
    int capacity;
}PriorityQueue;

// 向上调整,position表示当前调整开始的位置,范围是 0 ~ size - 1
void adjustUp(PriorityQueue *queue,int position){
    if(queue == NULL || position <= 0)
        return;
    int tempPriority = queue->unit[position].priority;
    DATATYPE tempData = queue->unit[position].data;

    for(int i = (position + 1) / 2 - 1;i >= 0;i = (i + 1) / 2 -1){
        if(queue->unit[i].priority > tempPriority)
            return;
        queue->unit[position].priority = queue->unit[i].priority;
        queue->unit[position].data = queue->unit[i].data;
        position = i;
    }
    queue->unit[position].priority = tempPriority;
    queue->unit[position].data = tempData;
}

// 向下调整,position表示当前调整开始的位置,范围是 0 ~ size - 1
```

```
void adjustDown(PriorityQueue *queue,int position,int size){
    if(queue == NULL || position < 0 || position >= size)
        return;
    int tempPriority = queue->unit[position].priority;
    DATATYPE tempData = queue->unit[position].data;
    for(int i = 2 * position + 1;i < size;i = i * 2 + 1){
        if(i + 1 < size && queue->unit[i].priority < queue->unit[i+1].prior
ity)
            i++;
        if(queue->unit[i].priority > tempPriority){
            queue->unit[position].priority = queue->unit[i].priority;
            queue->unit[position].data = queue->unit[i].data;
            position = i;
        }
    }
    queue->unit[position].data = tempData;
    queue->unit[position].priority = tempPriority;

}

// 优先级队列初始化
PriorityQueue* initialize(int size){
    if(size <= 0)
        return NULL;
    PriorityQueue *queue = (PriorityQueue*)malloc(sizeof(PriorityQueue));
    if(queue == NULL)
        printf("初始化优先级队列错误\n");
    queue->size = 0;
    queue->capacity = size;
    queue->unit = (element*)malloc(sizeof(element) * size);
    if(queue->unit == NULL){
        printf("初始化优先级队列错误\n");
        free(queue);
        return NULL;
    }
    return queue;

}

// 销毁优先级队列
void destroy(PriorityQueue *queue){
    if(queue == NULL)
        return;

    free(queue->unit);
    free(queue);
```

```
    queue = NULL;
}

// 插入新元素
int insert(PriorityQueue *queue,int priority,DATATYPE data){

    if(queue == NULL || queue->size == queue->capacity){
        printf("插入失败\n");
        return 0;
    }

    element *ele = (element*)malloc(sizeof(element));
    if(ele == NULL){
        printf("插入失败\n");
        return 0;
    }

    // 插入队列
    ++(queue->size);
    queue->unit[queue->size - 1].priority = priority;
    queue->unit[queue->size - 1].data = data;
    // 重新调整为大顶堆
    adjustUp(queue,queue->size - 1);
    return 1;
}

// 获取优先级最高的元素
element* getHighest(PriorityQueue *queue){
    if(queue == NULL || queue->size == 0){
        printf("优先级队列为空\n");
        return NULL;
    }
    return &(queue->unit[0]);
}

// 删除优先级最高的元素
void deleteHighest(PriorityQueue *queue){
    if(queue == NULL || queue->size == 0){
        printf("优先级队列为空\n");
    }

    // 将堆顶元素交换到末尾
    queue->unit[0].priority = queue->unit[queue->size - 1].priority;
    queue->unit[0].data = queue->unit[queue->size - 1].data;
    queue->unit[queue->size - 1].priority = -2147483648;
```

```
    queue->unit[queue->size - 1].data = -2147483648;

    // 重新调整前面的元素
    adjustDown(queue,0,--(queue->size));
}

int main(){
    int count;
    printf("优先级队列元素个数:\n");
    scanf("%d",&count);
    if(count <= 0)
        return 0;
    PriorityQueue *queue = initialize(count);
    int tempPriority;
    int tempData;
    for(int i = 0;i < count;i++){
        printf("输入第%d个数据、优先级:\n",i+1);
        scanf("%d %d",&tempData,&tempPriority);
        insert(queue,tempPriority,tempData);
    }

    printf("删除前\n");
    for(int i = 0;i < queue->size ;i++){
        printf("%d %d\n",(queue->unit[i].data),(queue->unit[i].priority));
    }
    deleteHighest(queue);

    printf("删除后\n");
    for(int i = 0;i < queue->size;i++){
        printf("%d %d\n",(queue->unit[i].data),(queue->unit[i].priority));
    }

    return 0;
}
```

排序

冒泡排序

```
void bubbleSort(int arr[],int length){
    if(length <= 0 || !arr)
        return;
    // flag 修改标记，如果一轮完毕没有修改则说明基本有序，直接返回
    int temp,flag = 0;
    for(int i = 0;i < length - 1;i++){
        for(int j = 0;j < length - i - 1;j++){
            // 升序
            if(arr[j] < arr[j + 1]){
                temp = arr[j];
                arr[j] = arr[j+1];
                arr[j + 1] =  temp;
                flag = 1;
            }
        }
        if(flag == 0)
            return;
    }
}
```

选择排序

```
void selectSort(int arr[],int length){
    if(length <= 0 || !arr)
        return;

    int temp;
    for(int i = 0;i < length - 1;i++){
        int min = i;

        // 找到最小的位置
        for(int j =  i + 1;j < length;j++){
            if(arr[j] < arr[min])
                min = j;
```

```
        }
        // 将最小的位置和当前位置的元素交换
        if(min!=i){
            temp = arr[min];
            arr[min] = arr[i];
            arr[i] = temp;
        }
    }
}
```

插入排序

```
void insertSort(int arr[],int length){
    if(length <= 0 || !arr)
        return;

    // 从第2个元素开始插入
    for(int i = 1;i < length;i++){
        int temp = arr[i];
        int j;
        // 将当前元素前面所有比其大的元素后移
        for(j = i - 1;j >= 0 && arr[j] > temp;j--){
            arr[j + 1] = arr[j];
        }

        arr[j + 1] = temp;
    }
}
```

希尔排序

```
void shellSort(int arr[],int length){

    // 步长每次变为原来的1/2
    for(int step = length / 2;step >= 1;step /= 2){

        // 将每隔step个元素当做一个序列进行插入排序
        for(int i = step;i < length;i++){
            int temp = arr[i];
            int j;
```

```
            for(j = i - step;j >= 0 && arr[j] > temp;j-=step){
                arr[j + step] = arr[j];
            }
            arr[j + step] = temp;
        }
    }
}
```

归并排序

```
#define SIZE 20
void merge(int arr[],int low,int mid,int high){
    int temp[SIZE];
    int i  = low,k = 0,j = mid + 1;
    while(i <= mid && j <= high){
        if(arr[i] <= arr[j]){
            temp[k++] = arr[i++];
        }else{
            temp[k++] = arr[j++];
        }
    }

    while(j <= high){
        temp[k++] = arr[j++];
    }
    while(i <= mid){
        temp[k++] = arr[i++];
    }
    for(i = 0,j = low;i <= high - low;){
        arr[j++] = temp[i++];
    }
}

void mergeSort(int arr[],int low,int high){
    if(!arr || low >= high)
        return;
    int mid = (low + high) / 2;
    mergeSort(arr,low,mid);
    mergeSort(arr,mid + 1,high);
    merge(arr,low,mid,high);
}
```

堆排序

```
// 调整大根堆,now是当前需要调整的节点,len是数组长度
void adjustDown(int arr[],int now,int len){
    int temp = arr[now];
    for(int i = now * 2 + 1;i < len;i = i * 2 + 1){
        if(i + 1 < len && arr[i] < arr[i + 1]){
            i++;
        }
        if(arr[i] < temp)
            break;

        arr[now] = arr[i];
        now = i;
    }
    arr[now] = temp;
}
void heapSort(int arr[],int size){
    // 建立大根堆
    for(int i = size / 2 - 1;i >= 0;i --){
        adjustDown(arr,i,size);
    }

    // 将大根堆的第一个元素和交换到后面
    for(int i = 0;i < size;i++){
        int temp = arr[0];
        arr[0] = arr[size - i - 1];
        arr[size - i - 1] = temp;
        adjustDown(arr,0,size - i - 1);
    }
}
```

快速排序

```
int partion(int arr[],int low,int high){

    // 随机生成枢纽位置
    int pivotIndex = low + rand() % (high - low + 1);
    int temp = arr[pivotIndex];
    arr[pivotIndex] =  arr[low];
    arr[low] = temp;
```

```
    while(low < high){

        while(low < high && arr[high] >= temp)
            high--;
        arr[low] = arr[high];

        while(low < high && arr[low] <= temp)
            low++;
        arr[high] = arr[low];
    }

    arr[low] = temp;

    return low;

}

void quickSort(int arr[],int low,int high){
    if(low < high){
        int pivot = partion(arr,low,high);
        quickSort(arr,low,pivot - 1);
        quickSort(arr,pivot + 1,high);
    }
}
```

表达式树转化为前缀表达式

```
typedef struct Btree{
     char data[1];
     struct Btree *left,*right;
}Btree;

// 表达式树转化为中缀表达式
void btree2exp(Btree *root,int depth){
    if(!root)
        return;
    if(root->left == NULL && root->right == NULL){
        printf("%c",root->data[0]);
    }else{
        // 第一层最外面不需要括号
        if(depth != 1){
            printf("(");
        }
        // 左子树表达式
        btree2exp(root->left,depth+1);
        // 符号
        printf("%c",root->data[0]);
        // 右节点表达式
        btree2exp(root->right,depth+1);
        if(depth != 1){
            printf(")");
        }
    }


}
```

最差版本

```
// 你是产品经理，目前正在带领一个团队开发新的产品。不幸的是，你的产品的最新版本没有通过质量检测。由于每个版本都是基于之前的版本开发的，所以错误的版本之后的所有版本都是错的。

// 假设你有 n 个版本 [1, 2, ..., n]，你想找出导致之后所有版本出错的第一个错误的版本。

// 你可以通过调用 bool isBadVersion(version) 接口来判断版本号 version 是否在单元测试中出错。实现一个函数来查找第一个错误的版本。你应该尽量减少对调用 API 的

// Forward declaration of isBadVersion API.
bool isBadVersion(int version);

int firstBadVersion(int n) {
    unsigned int left = 1,right = n;
    while(left < right){
        unsigned int  mid = (left + right) / 2;
        if(isBadVersion(mid)){
            right = mid;
        }
        if(!isBadVersion(mid)){
            left = mid + 1;
        }
    }


    return left;
}
```

链表的交点

```
// 编写一个程序，找到两个单链表相交的起始节点。
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};

// 算法思想：两个链表如果出现交叉，那么该节点及其后面的节点都是公共节点
// 设长度分别为m,n(m >=n) 那么较长的链表就需要从m - n处开始与较短链表的
// 第一个节点开始比较，这样如果有相交的部分，那么一定会有一个位置，两个节点
// 指向同一个节点
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {

        // 两个链表的当前位置
        ListNode *tempA = headA,*tempB = headB;

        // 求A,B长度
        int lengthA = 0,lengthB = 0;

        while(tempA){
            lengthA++;
            tempA = tempA->next;
        }
        while(tempB){
            lengthB++;
            tempB = tempB->next;
        }

        // A比B长,A先移动 lengthA - lengthB
        if(lengthA > lengthB){
            tempA = headA;
            tempB = headB;

            for(int i = 0;i < lengthA - lengthB;i++){
                tempA = tempA->next;
            }
        }else{
            tempA = headA;
```

```
            tempB = headB;

            for(int i = 0; i < lengthB - lengthA;i++){
                tempB = tempB->next;
            }
        }

        while(tempB && tempA){
            if(tempB == tempA){
                return tempB;
            }
            tempB = tempB->next;
            tempA = tempA->next;
        }
        return NULL;
    }
};
```

链表是否有环

```
// 给定一个链表，判断链表中是否有环。

// 为了表示给定链表中的环，我们使用整数 pos 来表示链表尾连接到链表中的位置（索引从 0 开始）。 如果 pos 是 -1，则在该链表中没有环。

/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     struct ListNode *next;
 * };
 */

#include <stdio.h>
#include <stdbool.h>

struct ListNode {
      int val;
      struct ListNode *next;
};

bool hasCycle(struct ListNode *head) {

    // 空链表或者只有一个节点直接返回
    if(!head || !head->next){
        return false;
    }

    // 快指针速度是慢指针的两倍，这样两个指针如果指向了同一个节点则说明
    //  出现了环
    struct ListNode *fast = head;
    struct ListNode *slow = head;

    while(fast && fast->next){
        fast = fast->next->next;

        slow = slow->next;
        if(fast == slow)
            return true;
    }
```

```
    return false;
}
```

字母异位词

```c
// 给定两个字符串 s 和 t ，编写一个函数来判断 t 是否是 s 的字母异位词。

// 即两个字符串的字符和次数完全相同，仅仅是出现的顺序不同

// 你可以假设字符串只包含小写字母。

#include<stdio.h>
#include<stdbool.h>

bool isAnagram(char * s, char * t){

    // 键值对的映射，key为字母的ASCII码减去一个偏移量后的值，该偏移量恰好使得小写字母a映射到
    // 索引0.value为出现的次处，默认是0表示没有出现过。
    int map['z' - 'a' + 1] = {0};

    // 遍历字符串s,每次遍历到一个字符将其次数加1
    for(int i = 0;s[i] != '\0';i++){
        map[s[i] - 'a']++;
    }

    // 遍历字符串t,每次 遍历到一个字符将其次数减1
    for(int j = 0;t[j] != '\0';j++){
        map[t[j] - 'a']--;

        // t中出现了s中不存在的字符，导致该字符出现此处为负，这种情况也不是异位词
        if(j < 'z' - 'a' + 1 && map[j] < 0)
            return false;
    }

    // 如果两个词是字母异位词，经过加减后所有的字母次数应该为0
    for(int k = 0;k < 'z' - 'a' + 1;k++){
        if(map[k] != 0)
            return false;
    }

    return true;

}
```

```c
int main(){

    printf("%d\n", isAnagram("rat","car"));

    return 0;
}
```

有效的数独

```c
// 判断一个 9x9 的数独是否有效。只需要根据以下规则，验证已经填入的数字是否有效即可。
// 数字 1-9 在每一行只能出现一次。
// 数字 1-9 在每一列只能出现一次。
// 数字 1-9 在每一个以粗实线分隔的 3x3 宫内只能出现一次。
// 参考：https://leetcode-cn.com/problems/valid-sudoku/solution/you-xiao-de-shu-du-by-leetcode/
#include<stdio.h>

int isValidSudoku(char board[][9], int boardSize, int* boardColSize){

    int colFlag[9][9] = {0};
    int rowFlag[9][9] = {0};
    int cellFlag[9][9] = {0};

    for(int i = 0;i < boardSize;i++){
        for(int j = 0; j < *boardColSize;j++){
            if(board[i][j] >= '1' && board[i][j] <= '9'){
                int num = board[i][j] - '0';
                if(rowFlag[i][num - 1] || colFlag[j][num - 1] || cellFlag[(i / 3) * 3 + j / 3][num - 1]){
                    printf("%d %d %d\n",rowFlag[i][num - 1],colFlag[j][num - 1],cellFlag[(i / 3) * 3 + j / 3][num - 1]);
                    return 0;
                }

                rowFlag[i][num - 1] = 1;
                colFlag[j][num - 1] = 1;
                cellFlag[(i / 3) * 3 + j / 3][num - 1] = 1;

            }
        }
    }

    return 1;
}


int main(){

    char arr[9][9] = {'8','3','.','.','7','.','.','.','.','6','.','.','1','
```

```
9','5','.','.','.','.','9','8','.','.','.','.','6','.','8','.','.','.','6',
'.','.','.','3','4','.','.','8','.','3','.','.','1','7','.','.','.','2','.'
,'.','.','6','.','6','.','.','.','.','.','2','8','.','.','.','.','4','1','9','.
','.','5','.','.','.','.','8','.','.','7','9'};

    int col = 9;

   int ret = isValidSudoku(arr,9,&col);

   printf("%d",ret);

    return 0;
}
```

数字阶数求和

```
// 数字阶数求和
#include<stdio.h>
int main(){
    int n,a;

    int arr[200] =  { 0 };  // 用于记录每个位

    scanf("%d %d",&a,&n);

    // 记录低位产生的进位信息
    int carry = 0;

    // 是否产生了进位，用于在进位导致位置可能超过n时判断当前位置是否需要加一
    int hasChanged = 0;

    // 记录当前位置
    int now = 0;

    // 当前位置不一定是局限于n,当数字很大时进位可能导致位置超过n
    for(int i = 0; i <= now;i ++){

        // 未产生进位之前的各位之和
        int base = (n - now >= 0) ? n - now : 0;

        // 原本值加上进位
        arr[i] = a * base + carry;

        // 修改进位
        carry = (arr[i] >= 10)?arr[i] / 10 : 0;

        // 是否产生了进位，因为需要记录上一步的进位，所以不能通过进位本身来判断
        hasChanged = (arr[i] >= 10) ? 1 : 0;

        // 产生进位之后剩下的值作为该位的值
        arr[i] = (arr[i] >= 10)?arr[i] % 10 : arr[i];

        // 当前位置加一的条件是：在 0 - (n - 2)之间一定会加一。或者当前位置大于 n - 1(即最高位)
        // 需要借助当前位的进位，如果向前产生了进位则加一
        if(i < n - 1 || (i >= n - 1 && hasChanged == 1)){
```

```
            now++;
        }

        // 每次循环完毕后将值还原
        hasChanged = 0;
    }

    // 低位是从0开始的，所以反向输出
    for(int i = now;i >= 0;i --){
        printf("%d",arr[i]);
    }

    printf("\n");

    return 0;
}
```

最长公共前缀

```
// 编写一个函数来查找字符串数组中的最长公共前缀。

// 如果不存在公共前缀，返回空字符串 ""。

#include <stdio.h>
#include <string.h>
#include <stdlib.h>

char * longestCommonPrefix(char **strs, int strsSize){
    if(strsSize == 0)
        return "";
    // 最长公共前缀的长度,初始时假设第一个串为最长前缀
    int max_common_len = strlen(strs[0]);

    // 第一个串为空串直接返回
    if(max_common_len == 0)
        return "";

    char *max_common_prefix = (char*)malloc(sizeof(char) * (max_common_len
+ 1));

    // i表示第几个串，将每个串和第一个串的每个字符进行比较
    for(int i = 1;i < strsSize;i++){

        // 只要有一个空串就直接返回
        if(strlen(strs[i]) == 0)
            return "";

        // max_common_len需要进行不断更新，每个遇到一个不相等的字符时就将该字符的位置
        // 作为新的长度
        for(int j = 0;j < max_common_len && strs[i][j] != '\0';){
            // 相等则比较下一个
            if(strs[i][j] == strs[0][j]){
                j++;
            }

            // 不相等则更新长度并比较下一个字符串
            if(strs[i][j] == '\0' || j == max_common_len || strs[i][j] != s
trs[0][j]){
                max_common_len = j;
```

```
                break;
            }
        }


    }


    if(max_common_len == 0)
        return "";


    strncpy(max_common_prefix,strs[0],max_common_len);
    max_common_prefix[max_common_len] = '\0';

    return max_common_prefix;

}

int main(){

    char arr[3][10] = {"flower","flow","flight"};

    printf("%s",longestCommonPrefix(arr,3));

    return 0;
}
```

最大利润二

```
// 给定一个数组，它的第 i 个元素是一支给定股票第 i 天的价格。

// 如果你最多只允许完成一笔交易（即买入和卖出一支股票），设计一个算法来计算你所能获取的最大利润。

// 注意你不能在买入股票前卖出股票。

// 动态规划


#include <stdio.h>

#include <stdlib.h>


int maxProfit(int* prices, int pricesSize){


    if(pricesSize <= 0)
        return 0;

    // 股票的变化值数组
    int *changes = (int *)malloc(sizeof(int) * (pricesSize - 1));

    for(int i = 1;i < pricesSize;i++){
        changes[i - 1] = prices[i] - prices[i - 1];
    }

    int temp = 0;

    // 用于记录历史出现的最大利润
    int max = 0;

    for(int j = 0;j < pricesSize - 1;j++){
        if(temp + changes[j] >= 0){
            temp += changes[j];
        }else{
            temp = 0;
        }
```

```
        max = (max > temp ? max : temp);
    }

    return max;


}



int main(){
    return 0;
}
```

寻找缺失的数

```c
// 给定一个包含 0, 1, 2, ..., n 中 n 个数的序列，找出 0 .. n 中没有出现在序列中的那个数。

#include <stdio.h>

// 用[0 - numsSize] 之间的每一个元素和指定的数组进行异或运算
// 最后的元素就是缺失的
int missingNumber(int* nums, int numsSize){
    int missingNum = 0;
    for(int i = 0;i < numsSize;i++){
        missingNum =  missingNum ^ nums[i] ^ i;
    }
    return missingNum ^ numsSize;
}

int main(){
    int arr[] = {3,0,1};
    printf("%d",missingNumber(arr,3));
    return 0;
}
```

将字符串转化为数字

```
// 请你来实现一个 atoi 函数，使其能将字符串转换成整数。

// 首先，该函数会根据需要丢弃无用的开头空格字符，直到寻找到第一个非空格的字符为止。

// 当我们寻找到的第一个非空字符为正或者负号时，则将该符号与之后面尽可能多的连续数字组合起来，作为该整数的正负号；假如第一个非空字符是数字，则直接将其与之后连续的数字字符组合起来，形成整数。

// 该字符串除了有效的整数部分之后也可能会存在多余的字符，这些字符可以被忽略，它们对于函数不应该造成影响。

// 注意：假如该字符串中的第一个非空格字符不是一个有效整数字符、字符串为空或字符串仅包含空白字符时，则你的函数不需要进行转换。

// 在任何情况下，若函数不能进行有效的转换时，请返回 0。

#include<stdio.h>
#include<ctype.h>
#include<limits.h>

int myAtoi(char * str){

    // 处理字符的当前位置
    int i = 0;

    long result = 0;

    // 符号，1表示正数
    int flag = 1;

    // 跳过开头的空格字符
    for(;str[i] != '\0' && str[i] == ' ';i++){

    }

    //  负号
    if(str[i] == '-' || str[i] == '+'){
        flag = str[i] == '-' ? 0 : 1;
        i++;
    }
```

```
    //  处理字符
    long  newResult;
    for(;str[i] != '\0' && isdigit(str[i]);i++){
        newResult =  result * 10 + (str[i] - '0');
        if(flag == 1 && newResult > INT_MAX){
            return INT_MAX;
        }

        if(flag == 0 && newResult * -1 < INT_MIN){
            return INT_MIN;
        }
        result = newResult;
    }

    // 带上符号
    if(flag == 0){
        result = -1 * result;
    }

    return (int)result;

}

int main(){
    char arr[100];

    scanf("%s",arr);

    printf("%d\n",myAtoi(arr));

    return 0;
}
```

抢劫--动态规划

```
// 你是一个专业的小偷，计划偷窃沿街的房屋。每间房内都藏有一定的现金，影响你偷窃的唯一制约因素就是

// 相邻的房屋装有相互连通的防盗系统，如果两间相邻的房屋在同一晚上被小偷闯入，系统会自动报警。

// 给定一个代表每个房屋存放金额的非负整数数组，计算你在不触动警报装置的情况下，能够偷窃到的最高金额。

#include <stdio.h>
#include <stdlib.h>

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

int rob(int* nums, int numsSize){

    // 没有房屋
    if(!nums || numsSize == 0)
        return 0;

    //  只有一家
    if(numsSize == 1)
        return nums[0];

    // 只有两家取较大者
    if(numsSize == 2)
        return max(nums[0],nums[1]);

    // 两家以上
    // dp[i]表示盗窃到第i家的时候能盗取的最大价值:
    // 1. 盗取第i家时,当前价值是dp[i - 2] + nums[i]
    // 2. 不盗取第i家时最大价值是dp[i - 1]
    // 应该在两者之间去较大值
    int *dp = (int*)malloc(sizeof(int) * numsSize);
    dp[0] = nums[0];
    dp[1] = max(nums[0],nums[1]);

    for(int i  = 2;i < numsSize;i++){
```

```
        dp[i] = max(dp[i - 2] + nums[i],dp[i - 1]);
    }

    return dp[numsSize - 1];

}
```

只出现一次的数字

```
// 给定一个非空整数数组，除了某个元素只出现一次以外，其余每个元素均出现两次。找出那个只出现了一次的元素。
// 你的算法应该具有线性时间复杂度。 你可以不使用额外空间来实现吗?

#include<stdio.h>

int singleNumber(int* nums, int numsSize){
    int result = 0;
    for(int i = 0; i < numsSize;i ++){
        result^=nums[i];
    }
    return result;

}

int main(){

    int arr[] = {4,1,2,1,2};

    int result = singleNumber(arr,5);

    printf("%d",result);



    return 0;
}
```

两个链表之和

```
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
// 给出两个 非空 的链表用来表示两个非负的整数。其中，它们各自的位数是按照 逆序
// 的方式存储的，并且它们的每个节点只能存储 一位 数字。
// 如果，我们将这两个数相加起来，则会返回一个新的链表来表示它们的和。

// 您可以假设除了数字 0 之外，这两个数都不会以 0 开头。
struct ListNode {
   int val;
   ListNode *next;
   ListNode(int x) : val(x), next(NULL) {}
};
// 思想:两个链表从低索引到高索引对应数的低位到高位，所以从开头开始计算，其结果就是低位到高位的计算
// 需要注意的是可能会产生进位，所以需要一个进位位,这个位要加到下一个位的计算上。
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {

        // 新生成的链表的头指针
        ListNode *head = NULL;

        // p1,p2分别表示两个链表的当前位置,p表示生成链表的当前位置
        ListNode *p1 = l1,*p2 = l2,*p = head;

        // temp暂存当前位的和
        int temp;

        // 进位位
        int carry = 0;

        while(p1 != NULL && p2 != NULL){
            // 两个位的初始和，这个和可能会超过10
            temp = p1->val + p2->val + carry;
```

```
            // 产生进位的情况
            if(temp > 9){
                carry = temp / 10;
                temp %= 10;
            }

            // 没有进位时进位位归零
            else{
                carry =0;
            }

            // 第一次计算没有头结点，从头结点开始连接
            if(head == NULL){
                head = new ListNode(temp);
                p = head;
            }
            else{
                p->next = new ListNode(temp);
                p = p->next;
            }
            p1 = p1->next;
            p2 = p2->next;
        }

        // 两个链表长度不等时，直接将较长链表的位和carry做加法
        while(p1!=NULL){
            temp = p1->val + carry;
            if(temp > 9){
                carry = temp / 10;
                temp %= 10;
            }else{
                carry =0;
            }
            p->next = new ListNode(temp);
            p = p->next;
            p1 = p1->next;
        }

        while(p2!=NULL){
            temp = p2->val + carry;
            if(temp > 9){
                carry = temp / 10;
                temp %= 10;
            }else{
                carry =0;
            }
```

```
            p->next = new ListNode(temp);
            p = p->next;
            p2 = p2->next;
        }

        // 最高位之和如果产生了进位，那么需要额外的一个节点保存其值
        if(carry != 0){
            p->next = new ListNode(carry);
        }

        return head;

    }
};
```

合法的二叉搜索树

```java
/**
 * Definition for a binary tree node.
 * public class TreeNode {
 *     int val;
 *     TreeNode left;
 *     TreeNode right;
 *     TreeNode(int x) { val = x; }
 * }
 */
class Solution {
    public boolean isValidBST(TreeNode root) {

        // 保存中序遍历的值
        List<Integer> list = new ArrayList();

        if(root == null)
            return true;

        // 当前位置
        TreeNode pos = root;

        // 辅助栈
        Stack<TreeNode> stack = new Stack();

        while(pos != null || !stack.isEmpty()){

            // 向左访问，直到到达最左边的叶子节点
            if(pos != null){
                stack.push(pos);
                pos = pos.left;
            }
            // 将最左节点出栈并开始操作该节点的右子树
            else{
                TreeNode node = stack.pop();

                list.add(node.val);

                pos = node.right;

            }
```

```
        }

        // 二叉搜索树的中序序列必然是递增的
        for(int i = 0;i < list.size() - 1;i++){
            if(list.get(i) >= list.get(i + 1))
                return false;
        }
        return true;
    }
}
```

寻找第一个匹配的子字符串

```
//  实现 strStr() 函数。

// 给定一个 haystack 字符串和一个 needle 字符串，

// 在 haystack 字符串中找出 needle 字符串出现的第一个位置 (从0开始)。如果不存在，则返回  -1。

#include<stdio.h>
#include<stdlib.h>

// 采用 kmp 方式匹配
int strStr(char * haystack, char * needle){

    // needle的长度
    int len_needle = 0;
    for(;needle[len_needle] != '\0';len_needle++){

    }

    // 如果 needle 是空串直接返回0
    if(len_needle == 0)
        return 0;

    // next数组
    int  *next = (int*)malloc(sizeof(int) * len_needle);

    // 填充next数组
    next[0] = -1;
    for(int j = 0,k =  -1;j < len_needle - 1;){
        if( k ==  -1  || needle[j] == needle[k]){
            next[++j] =  ++k;
        }else{
            k = next[k];
        }
    }

    //  kmp模式匹配
    int i = 0,j = 0;
    // 这里不能用 needle[j] != '\0' 判断，因为j可能会回溯到 -1，这样对数组进行操作可能
```

```
    // 会出现问题
    for(;haystack[i] != '\0' && j < len_needle;){
        if(j == -1 || haystack[i] == needle[j]){
            i++;
            j++;
        }else{
            j = next[j];
        }
    }

    if(j + 1 > len_needle){
        return i - j;
    }else{
        return -1;
    }


}


int main(){

    char str1[100],str2[100];
    scanf("%s %s",str1,str2);
    printf("%d",strStr(str1,str2));
    return 0;
}
```