# 难点攻克

1单调栈

2广度优先搜索

3并查集

4啥是迭代，与递归分清

5异或运算

6.打家劫舍系列问题，爬楼梯系列问题

# 小优化

1. =[--P假如求中位数可以 (l+r)>>1;
2. vector容器插入操作用emplace\_back();
3. 尽可能用前置加减代替后置加减。

# 模板和套路

1. 对于有序数组，可以使用二分查找的方法查找元素。二分查找时间复杂度为O(logn).
2. 对于这类寻找所有可行解的题，我们都可以尝试用「搜索回溯」的方法来解决。
3. 回溯法:一种通过探索所有可能的候选解来找出所有的解的算法。如果候选解被确认不是一个解(或者至少不是最后- -个解)，回溯算法会通过在上一步进行一些变化抛弃该解，即回溯并且再次尝试。

回溯算法用于寻找所有的可行解，如果发现一个解不可行，则会舍弃不可行的解。

1. 检查一个字符串是否出现在给定的字符串列表里一般可以考虑哈希表来快速判断
2. 一旦涉及出现次数，可以使用哈希表
3. **判断是否有重复字符，可以使用哈希数据结构。**
4. 如果求可行性解，给定一个有重复数字的序列，记住要先排序，然后再回溯，设置剪枝条件
5. 链表求长度或第k个元素，可以通过双指针来解决
6. 二叉搜索树的中序遍历是递增的。
7. 求二叉树的深度可以用层序遍历，遍历一层，层数+1；
8. 这类链表题目一般都是使用双指针法解决的，例如寻找距离尾部第K个节点、寻找环入口、寻找公共尾部入口等。
9. 若是有环链表，快慢指针指向头结点，快指针走两步，满指针走一步两指针相遇，让快指针指向头结点，然后快慢指针都一步一步的走，相遇的点就是

# 两数之和：

方法一：暴力求解，遍历nums中的每一个元素，两遍for，令nums[i]+nums[j]=target时返回[i,j];

class Solution {

public:

vector<int> twoSum(vector<int>& nums, int target) {

int i=0,j=0;

for(i=0;i<nums.size();i++)

{

for(j=i+1;j<nums.size();j++)

{

if(nums[i]+nums[j]==target)

{

return {i,j};

}

}

}

return {i,j};

}

};

方法二：借用哈希表，时间复杂度0（n）,将元素存储在哈希表中，在哈希表中查找target-nums[i]的时间复杂度为常数复杂度，

class Solution {

public:

vector<int> twoSum(vector<int>& nums, int target) {

unordered\_map<int,int> hashtable;

for(int i=0;i<nums.size();++i)

{

auto it=hashtable.find(target-nums[i]);

if(it!=hashtable.end())

{

return {it->second,i};

}

hashtable[nums[i]]=i;

}

return {};

}

};

# 两数相加

先定义一个辅助结点存储两数之和，辅助指针指向该节点。遍历L1和L2，将L1和L2的值赋给N1，N2。Sum=N1+N2+carry。指针指向下一个创建结点并赋值sum%10。最后carry如果大于0，指针在创建一个结点放carry。返回辅助结点的下一个结点。

别忘了最后的进位。

class Solution {

public:

ListNode\* addTwoNumbers(ListNode\* l1, ListNode\* l2) {

ListNode\*head=new ListNode(0);

ListNode\*cur=head;

int carry=0;

while(l1||l2)

{

int sum=0;

if(l1) sum+=l1->val;

if(l2) sum+=l2->val;

sum+=carry;

carry=sum/10;

cur->next=new ListNode(sum%10);

cur=cur->next;

if(l1) l1=l1->next;

if(l2) l2=l2->next;

}

if(carry) cur->next=new ListNode(carry);

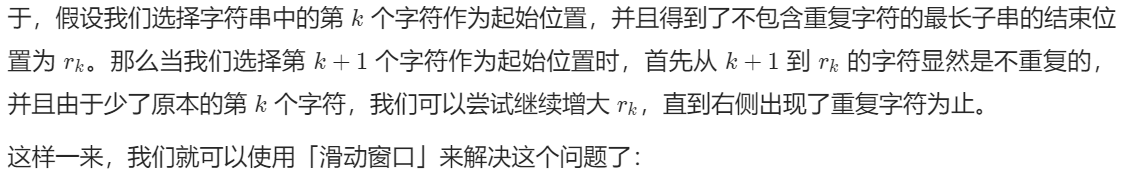
return head->next;

}

};

# 无重复字符的最长子串

滑动窗口，



定义一个哈希集合occ，遍历lk左指针，如果左指针向右移动一位，就erase第一个元素 ，while循环右移指针，给occ 插入元素，更新ans=max(ans,rk-i);

class Solution {

public:

int lengthOfLongestSubstring(string s) {

unordered\_set<char> ans;// 哈希集合，记录每个字符是否出现过

int maxcount=0;

int rk=0;

for(int i=0;i<s.size();i++)// 枚举左指针的位置，

{

if(i!=0) ans.erase(s[i-1]);// 左指针向右移动一格，移除一个字符

while(rk<s.size()&&!ans.count(s[rk]))// 不断地移动右指针

{

ans.insert(s[rk]);

++rk;

}

maxcount=max(rk-i,maxcount);// 第 i 到 rk 个字符是一个极长的无重复字符子串

}

return maxcount;

}

};

# 寻找两个正序数组的中位数：

就是在两个有序数组中找第k小的数。

根据中位数的定义，当 m+n 是奇数时，中位数是两个有序数组中的第 (m+n)/2+1 个元素，当 m+n 是偶数时，中位数是两个有序数组中的第 (m+n)/2 个元素和第 (m+n)/2+1 个元素的平均值。因此，这道题可以转化成寻找两个有序数组中的第 k 小的数，其中 k 为 (m+n)/2 或 (m+n)/2+1。

假设两个有序数组分别是A和B。要找到第k个元素，我们可以比较A[k/2-1]和B[k/2-1],其中/表示整数除法。由于A[k/2- 1]和B[k/2 - 1]的前面分别有A[0..k/2- 2] 和B[0..k/2-2]，即k/2- 1个元素，对于A[k/2- 1]和B[k/2 - 1]中的较小值，最多只会有(k/2-1)+ (k:/2- 1)≤k-2个元素比

它小，那么它就不能是第k小的数了。

因此我们可以归纳出三种情况:

●如果A[k/2-1]< B[k:/2- 1]， 则比A[k/2- 1]小的数最多只有A的前k/2- 1个数和B的前

k:/2-1个数,即比A[k/2 - 1] 小的数最多只有k-2个，因此A[k/2 - 1]不可能是第k个数, A[0]

到A[k:/2 - 1]也都不可能是第k个数，可以全部排除。

●如果A[k/2- 1]> B[:/2 - 1],则可以排除B[0]到B[k/2 - 1]。

●如果A[k/2- 1]= B[k:/2- 1]， 则可以归入第一-种情况处理。

可以看到，比较A[k/2- 1] 和B[k/2- 1]之后,可以排除k:/2 个不可能是第k小的数，查找范围缩小了

一半。同时，我们将在排除后的新数组上继续进行二分查找，并且根据我们排除数的个数，减少k的值,

这是因为我们排除的数都不大于第k小的数。

有以下三种情况需要特殊处理:

●如果A[k/2- 1]或者B[k/2- 1]越界，那么我们可以选取对应数组中的最后一个元素。在这种情况下，我们必须根据排除数的个数减少k的值，而不能直接将k减去k/2。

●如果一个数组为空，说明该数组中的所有元素都被排除，我们可以直接返回另一个数组中第k小的元素。

●如果k=1，我们只要返回两个数组首元素的最小值即可。

代码：

class Solution {

public:

int getKthElement(const vector<int>& nums1,const vector<int>& nums2,int k)

{

int m=nums1.size();

int n=nums2.size();

int index1=0,index2=0;

while(true)

{//边界情况

//1.如果一个数组为空，说明该数组中的所有元素都被排除，我们可以直接返回另一个数组中第k小的元素

if(index1==m) return nums2[index2+k-1];

if(index2==n) return nums1[index1+k-1];

//2.如果k=1，我们只要返回两个数组首元素的最小值即可。

if(k==1) return min(nums1[index1],nums2[index2]);

//正常情况

int newIndex1=min(index1+k/2-1,m-1);

int newIndex2=min(index2+k/2-1,n-1);

int pivot1=nums1[newIndex1];

int pivot2=nums2[newIndex2];

if(pivot1<=pivot2)

{

//如果A[k/2- 1]或者B[k/2- 1]越界，那么我们可以选取对应数组中的最后一个元素。

//在这种情况下， 我们必须根据排除数的个数减少k的值，而不能直接将k减去k/2。

k=k-(newIndex1+1-index1);

index1=newIndex1+1;

}

else

{

k=k-(newIndex2+1-index2);

index2=newIndex2+1;

}

}

}

double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {

int totalLength=nums1.size()+nums2.size();

if(totalLength%2==1)

{

return getKthElement(nums1,nums2,(totalLength+1)/2);

}

else

{

return (getKthElement(nums1,nums2,totalLength/2)+getKthElement(nums1,nums2,totalLength/2+1))/2.0;

}

}

};

# 最长回文子串：

中心扩展法：

class Solution {

public:

string longestPalindrome(string s) {

string ans="";

for(int i=0;i<s.size();i++)

{

//如果回文串为奇数

int l=i-1,r=i+1;

while(l>=0&&r<s.size()&&s[l]==s[r]) l--,r++;

if(ans.size()<r-l-1) ans=s.substr(l+1,r-l-1); //左右两部分都多扩展了一位,l-r+1-2

//如果回文串为偶数

l=i,r=i+1;

while(l>=0&&r<s.size()&&s[l]==s[r]) l--,r++;

if(ans.size()<r-l-1) ans=s.substr(l+1,r-l-1);

}

return ans;

}

};

方法二:动态规划

回文天然具有「状态转移」性质:一个长度严格大于2的回文去掉头尾字符以后，剩下的部分依然是回文。反之,如果一个字符串头尾两个字符都不相等，那么这个字符串一定不是回文。「动态规划」的方法

根据这样的性质得到。

第1步:定义状态

dp[i][j]表示:子串s[i..j]是否为回文子串，这里子串s[i.. j]定义为左闭右闭区间，即可以取到s[i]和s[j]。

第2步:思考状态转移方程根据头尾字符是否相等，需要分类讨论

dp[i][j] = (s[i] == s[j]) and dp[i + 1][j-1]

说明:

●「动态规划」的「自底向上」求解问题的思路，很多时候是在填写一-张二维表格。由于s[i..j]表示s的一个子串，因此i和j的关系是i <= j，只需要填这张表格对角线以上的部分;

●看到dp[i + 1][j- 1]就需要考虑特殊情况:如果去掉s[i.. j] 头尾两个字符子串s[i+ 1,j-1]

的长度严格小于2 (不构成区间)，即j- 1-(i+1)+1<2时，整理得j-i< 3,此时s[i..j] 是否是回文只取决于s[i]与s[j]是否相等。结论也比较直观: j-i< 3等价于j-i+1<4，即当子串s[i..j] 的长度等于2或者等于3的时候，s[i..j] 是否是回文由s[i] 与s[j] 是否相等决定。

第3步:考虑初始化

单个字符一定是回文串，因此把对角线先初始化为true ，即dp[i][i] = true。根据第2步的说明:当s[i..j] 的长度为2时，只需要判断s[i]是否等于s[j] ，所以二维表格对角线上的数值不会被参考。所以不设置dp[i][i] = true 也能得到正确结论。

第4步:考虑输出

一旦得到dp[i][j] = true ，就记录子串的「长度」和「起始位置」。没有必要截取,这是因为截取字符串也有性能消耗。

第5步:考虑优化空间

●下面给出的「参考代码」，在填表的过程中，只参考了左下方的数值。事实上可以优化,但是增加了代码编写和理解的难度,丢失了可读性和可解释性。在这里不做优化空间;

●填表应该遵守这样的原则:总是先得到小子串是否是回文的结果，然后大子串才能参考小子串的判断结果，所以填表顺序很重要;

代码：

class Solution {

public:

string longestPalindrome(string s) {

int n=s.size();

if(n<2) return s;

int maxlen=1;

int begin=0;

//// dp[l][r] 表示 s[l..r] 是否是回文串

vector<vector<int>> dp(n,vector<int>(n));

//初始化，所有长度为1的子串都是回文串

for(int i=0;i<n;i++) dp[i][i]=1;

for(int r=1;r<n;r++)//枚举右指针

{

for(int l=0;l<r;l++)//枚举左指针

{

if(s[l]!=s[r])

{

dp[l][r]=0;

}

else

{

if(r-l<3)//文回串长度为奇数可以是3，l-r+1<4或(r-1)-(l+1)+1<2

{

dp[l][r]=1;

}

else//回文串长度为偶数

{

dp[l][r]=dp[l+1][r-1];

}

}

//只要dp[l][r]为1，就表示子串s[l...l]为回文，此时记录回文长度和起始位置

if(dp[l][r]&&r-l+1>maxlen)

{

maxlen=r-l+1;

begin=l;

}

}

}

return s.substr(begin,maxlen);

}

};

1. Z字形变换：创建一个vector数组，在bool一个方向，遍历。
2. 1
3. 1
4. 1

# 正则表达式

看看题解，理解意思

# 盛最多水的容器：

双指针解法，while(l<r),,求容积，如果nums【r】<nums[l],r++;

代码：class Solution {

public:

int maxArea(vector<int>& height) {

int ans=0;

int l=0,r=height.size()-1;

while(l<r)

{

ans=max(ans,min(height[l],height[r])\*(r-l));

if(height[l]<height[r]) l++;

else r--;

}

return ans;

}

};

1. 颜色分类：定义三个指针i=0,j=0,k=nums.size()-1.遍历j指针，若nus[j]==0,交换 nums[i],nums[j]，j++,i++,若nums[j]==2,交换nums[j],nums[k],k--,否则j++;

# 三数之和

方法：排序+双指针法

●标签:数组遍历

●首先对数组进行排序，排序后固定一个数nums[i],再使用左右指针指向nums[i]后面的两端，数字

分别为nums[]和nums[R],计算三个数的和sum判断是否满足为0,满足则添加进结果集

●如果nums[i]大于0,则三数之和必然无法等于0，结束循环

●如果nums[i] == nums[i- 1],则说明该数字重复，会导致结果重复，所以应该跳过

●当sum==0时，nums[] == nums[[+ 1]则会导致结果重复，应该跳过，L++

●当sum==0时，nums[R] == nums[R- 1]则会导致结果重复，应该跳过, R- -

时间复杂度: O(n2), n为数组长度

代码：class Solution {

public:

vector<vector<int>> threeSum(vector<int>& nums) {

int n=nums.size();

sort(nums.begin(),nums.end());

vector<vector<int>> ans;

if(n==0) return ans;

If(nums[0]>0) return ans;

for(int i=0;i<n;++i)

{

int l=i+1,r=n-1;

int target=0-nums[i];

if(i>0&&nums[i]==nums[i-1]) continue;

while(l<r)

{

if(nums[l]+nums[r]==target)

{

ans.push\_back({nums[i],nums[l],nums[r]});

while(l<r&&nums[r]==nums[r-1]) --r;

while(l<r&&nums[l]==nums[l+1]) ++l;

--r;

++l;

}

else if(nums[l]+nums[r]<target)

{

++l;

}

else

{

--r;

}

}

}

return ans;

}

};

# 电话号码的字母组合

代码：dfs

class Solution {

public:

string strs[10]={"","","abc","def","ghi","jkl","mno","pqrs","tuv","wxyz"};

vector<string> ans;

vector<string> letterCombinations(string digits) {

if(digits.size()==0) return ans;

dfs(digits,"",0);

return ans;

}

void dfs(string digits,string combine,int idx)

{

if(idx==digits.size())

{

ans.push\_back(combine);

return ;

}

string s=strs[digits[idx]-'0'];

for(int i=0;i<s.size();i++)

{

combine.push\_back(s[i]);

dfs(digits,combine,idx+1);

combine.pop\_back();

}

}

};

# 删除链表第n个结点

方法一：思路与算法

- -种容易想到的方法是，我们首先从头节点开始对链表进行一次遍历，得到链表的长度L。随后我们再从

头节点开始对链表进行一次遍历， 当遍历到第L- n+ 1个节点时，它就是我们需要删除的节点。

为为与题目中的n保持-致，节点的编号从1开始，头节点为编号1的节点。

为了方便删除操作,我们可以从哑节点开始遍历L-n+1个节点。当遍历到第L-n+1个节点时，它

的下一个节点就是我们需要删除的节点，这样我们只需要修改一次指针， 就能完成删除操作。

代码:

class Solution {

public:

ListNode\* removeNthFromEnd(ListNode\* head, int n) {

ListNode\*dummy=new ListNode(0,head);

ListNode\*cur=dummy;

int count=0;

while(head)

{

count++;

head=head->next;

}

for(int i=1;i<count-n+1;++i)

{

cur=cur->next;

}

cur->next=cur->next->next;

ListNode\*ans=dummy->next;

delete dummy;

return ans;

}

};

方法二：

由于我们需要找到倒数第n个节点，因此我们可以使用两个指针first 和second同时对链表进行遍历，并

且first比second超前n个节点。当first 遍历到链表的末尾时，second 就恰好处于倒数第n个节点。

具体地，初始时first 和

second均指向头节点。我们首先使用first对链表进行遍历，遍历的次数为n。此

时，first 和second之间间隔了n- 1个节点,即first比second超前了n个节点。

在这之后，我们同时使用first 和second对链表进行遍历。当first 遍历到链表的末尾(即first 为空指

针)时，second恰好指向倒数第n个节点。

根据方法- -和方法二，如果我们能够得到的是倒数第n个节点的前驱节点而不是倒数第n个节点的话,

删除操作会更加方便。因此我们可以考虑在初始时将second指向哑节点，其余的操作步骤不变。这样一

来，当first 遍历到链表的末尾时，second 的下一个节点就是我们需要删除的节点。

代码：

class Solution {

public:

ListNode\* removeNthFromEnd(ListNode\* head, int n) {

ListNode\*dummy=new ListNode(0,head);

ListNode\*first=head;

ListNode\*second=dummy;

for(int i=0;i<n;i++)

{

first=first->next;

}

while(first)

{

first=first->next;

second=second->next;

}

second->next=second->next->next;

ListNode\*ans=dummy->next;

delete dummy;

return ans;

}

};

# 有效的括号

方法：栈

判断括号的有效性可以使用「栈」这一数据结构来解决。

我们遍历给定的字符串s。当我们遇到一个左括号时，我们会期望在后续的遍历中，有一一个相同类型的右

括号将其闭合。由于后遇到的左括号要先闭合，因此我们可以将这个左括号放入栈顶。

当我们遇到一一个右括号时，我们需要将一个相同类型的左括号闭合。 此时，我们可以取出栈顶的左括号并

判断它们是否是相同类型的括号。如果不是相同的类型，或者栈中并没有左括号，那么字符串s无效,返

回False。为了快速判断括号的类型，我们可以使用哈希表存储每一种括号。 哈希表的键为右括号,值为

相同类型的左括号。

在遍历结束后，如果栈中没有左括号，说明我们将字符串s中的所有左括号闭合，返回True,否则返回

F alse。

注意到有效字符串的长度一定为偶数， 因此如果字符串的长度为奇数，我们可以直接返回False,省去后

续的遍历判断过程。

代码：

class Solution {

public:

bool isValid(string s) {

unordered\_map<char,char> pairs={{')','('},{'}','{'},{']','['}};

stack<char> ans;

for(int i=0;i<s.size();++i)

{

if(pairs.count(s[i]))

{

if(ans.empty()||ans.top()!=pairs[s[i]]) return false;

ans.pop();

}

else

{

ans.push(s[i]);

}

}

return ans.empty();

}

};

# 合并两个有序数组

代码：

class Solution {

public:

ListNode\* mergeTwoLists(ListNode\* list1, ListNode\* list2) {

ListNode\*h3=new ListNode(0);

ListNode\*cur=h3;

while(list1&&list2)

{

if(list1->val<=list2->val)

{

cur->next=new ListNode(list1->val);

list1=list1->next;

}else

{

cur->next=new ListNode(list2->val);

list2=list2->next;

}

cur=cur->next;

}

cur->next=list1?list1:list2;

return h3->next;

}

};

[一文搞定常见的链表问题 (欢迎交流 - 环形链表 - 力扣（LeetCode）](https://leetcode.cn/problems/linked-list-cycle/solution/yi-wen-gao-ding-chang-jian-de-lian-biao-wen-ti-h-2/)

# 括号生成

方法:回溯法

思路和算法

我们可以通过跟踪到目前为止放置的左括号和右括号的数目来做到，

1.左括号和右括号数量相等为n.

2.如果左括号数量不大于n,我们可以放一一个左括号。如果右括号数量小于左括号的数量，我们可以放一个右括号。

代码：

class Solution {

public:

vector<string> ans;

string combine;

vector<string> generateParenthesis(int n) {

if(n==0) return ans;

dfs(n,0,0);

return ans;

}

void dfs(int n,int lc,int rc)

{

if(lc==n&&rc==n)

{

ans.push\_back(combine);

return ;

}

if(lc<n)

{

combine+="(";

dfs(n,lc+1,rc);

combine.pop\_back();

}

if(rc<lc)

{

combine+=")";

dfs(n,lc,rc+1);

combine.pop\_back();

}

}

};

# 合并K个升序链表

方法一：

思路：顺序合并，将两个有序链表合并。

代码：

class Solution {

public:

ListNode\* merge(ListNode\*list1,ListNode\*list2)

{

ListNode\*dummy=new ListNode(0);

ListNode\*cur=dummy;

while(list1&&list2)

{

if(list1->val<=list2->val)

{

cur->next=new ListNode(list1->val);

list1=list1->next;

}

else

{

cur->next=new ListNode(list2->val);

list2=list2->next;

}

cur=cur->next;

}

cur->next=list1?list1:list2;

return dummy->next;

}

ListNode\* mergeKLists(vector<ListNode\*>& lists) {

ListNode\*list=nullptr;

for(int i=0;i<lists.size();++i)

{

list=merge(list,lists[i]);

}

return list;

}

};

方法二：

分治合并

代码：

/\*\*

\* Definition for singly-linked list.

\* struct ListNode {

\* int val;

\* ListNode \*next;

\* ListNode() : val(0), next(nullptr) {}

\* ListNode(int x) : val(x), next(nullptr) {}

\* ListNode(int x, ListNode \*next) : val(x), next(next) {}

\* };

\*/

class Solution {

public:

ListNode\* merge(ListNode\*list1,ListNode\*list2)

{

ListNode\*dummy=new ListNode(0);

ListNode\*cur=dummy;

while(list1&&list2)

{

if(list1->val<=list2->val)

{

cur->next=new ListNode(list1->val);

list1=list1->next;

}

else

{

cur->next=new ListNode(list2->val);

list2=list2->next;

}

cur=cur->next;

}

cur->next=list1?list1:list2;

return dummy->next;

}

ListNode\* merge(vector <ListNode\*> &lists, int l, int r) {

if (l == r) return lists[l];

if (l > r) return nullptr;

int mid = (l + r) >> 1;

return merge(merge(lists, l, mid), merge(lists, mid + 1, r));

}

ListNode\* mergeKLists(vector<ListNode\*>& lists) {

return merge(lists, 0, lists.size() - 1);

}

};

# 下一个排列

方法：双扫描法

1.首先从后向前查找第一一个顺序对(i-1,i)，满足a[i-1]<a[i]。这样「较小数」即为a[i-1]。此时[i ,n)必然是下降序列。

2.如果找到了顺序对，那么在区间[i ,n)中从后向前查找第一个元素j满足a[i-1]<a[j]。这样「较

大数」即为a[j]。

1. 交换a[i-1]与a[j]， 此时可以证明区间[i ,n)必为降序。我们可以直接使用双指针反转区间

[i , n)使其变为升序，而无需对该区间进行排序。

****注意****

如果在步骤 1 找不到顺序对，说明当前序列已经是一个降序序列，即最大的序列，我们直接反转数组，即可得到最小的升序序列。

代码：

class Solution {

public:

void nextPermutation(vector<int>& nums) {

int i=nums.size()-1;

while(i>0&&nums[i-1]>=nums[i]) i--;

if(i<=0)

{

reverse(nums.begin(),nums.end());

}

else

{

int j=nums.size()-1;

while(nums[j]<=nums[i-1]) j--;

swap(nums[j],nums[i-1]);

reverse(nums.begin()+i,nums.end());

}

}

};

# 最长有效括号

[动态规划思路详解（c++）——32.最长有效括号 - 最长有效括号 - 力扣（LeetCode）](https://leetcode.cn/problems/longest-valid-parentheses/solution/dong-tai-gui-hua-si-lu-xiang-jie-c-by-zhanganan042/)

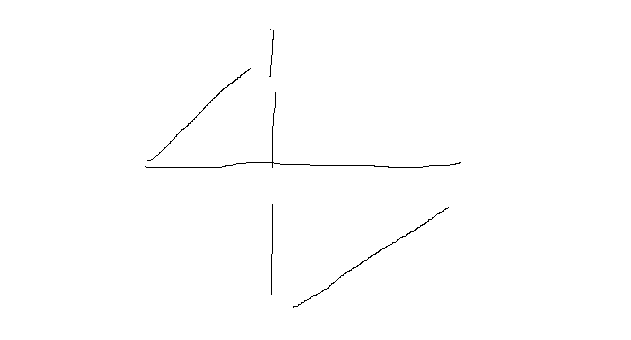
理解为主。想思路。

# 搜索旋转数组

思路：将数组一分为二，其中一定有一个是有序的，另一个可能是有序，也能是部分有序。此时有序部分用二分法查找。无序部分再一分为二，其中一个一定有序，另一个可能有序，可能无序。就这样循环.

●如果[1, mid- 1]是有序数组，且target 的大小满足[nums[l], nums[mid]), 则我们应该将搜索范围缩小至[1， mid-1]，否则在[mid + 1, r]中寻找。

●如果[mid, r]是有序数组，且target 的大小满足(nums[mid + 1]，nums[r]， 则我们应该将搜索范围缩小至[mid + 1，r]，否则在[1, mid-1]中寻找。



代码：

class Solution {

public:

int search(vector<int>& nums, int target) {

if(nums.size()==0) return -1;

int l=0,r=nums.size()-1;

while(l<=r)

{

int mid=(l+r)/2;

if(nums[mid]==target) return mid;

if(nums[0]<=nums[mid])

{

if(target>=nums[l]&&target<nums[mid])

{

r=mid-1;

}

else

{

l=mid+1;

}

}

else

{

if(target>nums[mid]&&target<=nums[r])

{

l=mid+1;

}

else

{

r=mid-1;

}

}

}

return -1;

}

};

# [34. 在排序数组中查找元素的第一个和最后一个位置](https://leetcode.cn/problems/find-first-and-last-position-of-element-in-sorted-array/)

思路：

[图解二分 | 最清晰易懂的讲解 | 一次性帮你解决二分边界问题【c++/java版本】 - 在排序数组中查找元素的第一个和最后一个位置 - 力扣（LeetCode）](https://leetcode.cn/problems/find-first-and-last-position-of-element-in-sorted-array/solution/tu-jie-er-fen-zui-qing-xi-yi-dong-de-jia-ddvc/)

代码：

class Solution {

public:

vector<int> searchRange(vector<int>& nums, int target) {

if(nums.empty()) return {-1,-1};

int l=0,r=nums.size()-1;//二分范围

while(l<r)//查找元素的开始位置

{

int mid=(l+r)>>1;

if(nums[mid]>=target) r=mid;

else l=mid+1;

}

if(nums[r]!=target) return {-1,-1};//查找失败

int L=r;

l=0,r=nums.size()-1;

while(l<r)//查找元素的结束位置

{

int mid=(l+r+1)>>1;

if(nums[mid]<=target) l=mid;

else r=mid-1;

}

return {L,l};

}

};

# 组合总和

对于这类寻找所有可行解的题，我们都可以尝试用「搜索回溯」的方法来解决。

代码：

class Solution {

public:

vector<vector<int>> ans;//结果容器

vector<int> combine;//组合容器

void dfs(vector<int>& candidates,int target,int sum,int idx)

{

if(sum==target)//根据题意，确立结束条件

{

ans.push\_back(combine);

return ;

}

for(int i=idx;i<candidates.size();++i)//找准选择列表(与函数参数相关),与第一步紧密关联

{

if(sum>target) continue;//判断是否需要剪枝

combine.push\_back(candidates[i]);//作出选择，递归调用，进入下一层

dfs(candidates,target,sum+candidates[i],i);

combine.pop\_back();撤销选择

}

}

vector<vector<int>> combinationSum(vector<int>& candidates, int target) {

if(candidates.empty()) return ans;

dfs(candidates,target,0,0);

return ans;

}

};

# 接雨水

思路：双指针，单调栈（看看，挺有意思）

对于下标i，下雨后水能到达的最大高度等于下标i两边的最大高度的最小值，下标i处能接的雨水量等于下标i处的水能到达的最大高度减去height[i]。

维护两个指针left和right,以及两个变量leftMax和rightMax，初始时left = 0, right= n-1, leftMax =0, rightMax= 0。指针left 只会向右移动，指针right 只会向左移动，在移动指针的过程中维护两个变量leftMax和rightMax的值。

当两个指针没有相遇时，进行如下操作:

●使用height[left]和height[right]的值更新leftMax和rightMax的值;

●如果. height[eft < height[right]， 则必有leftMax < rightMax， 下标left 处能接的雨水量等于leftMax - height[left], 将下标left处能接的雨水量加到能接的雨水总量,然后将left 加1 (即向右移;动一位) ;

●如果height[eft] ≥height[right]， 则必有leftMax ≥rightMax，下标right 处能接的雨水量等于

rightMax - height[right], 将下标right处能接的雨水量加到能接的雨水总量，然后将right 减1 (即向左移动一位)

当两个指针相遇时，即可得到能接的雨水总量。

代码：

class Solution {

public:

int trap(vector<int>& height) {

int l=0,r=height.size()-1,ans=0;

int lmax=0,rmax=0;

while(l<=r)

{

lmax=max(lmax,height[l]);

rmax=max(rmax,height[r]);

if(lmax<rmax)

{

ans+=lmax-height[l];

l++;

}

else

{

ans+=rmax-height[r];

r--;

}

}

return ans;

}

};

# 全排列

方法：回溯法

代码：

class Solution {

public:

vector<vector<int>> ans;

vector<int> combine;

vector<bool> used;

void dfs(vector<int>& nums,int idx)

{

if(combine.size()==nums.size())

{

ans.emplace\_back(combine);

return ;

}

for(int i=idx;i<nums.size();i++)

{

if(!used[i])

{

combine.emplace\_back(nums[i]);

used[i]=true;

dfs(nums,0);

combine.pop\_back();

used[i]=false;

}

}

}

vector<vector<int>> permute(vector<int>& nums) {

used=vector<bool>(nums.size(),false);

if(nums.empty()) return ans;

dfs(nums,0);

return ans;

}

};

# 旋转数组

方法：原地旋转数组，先对角线翻转，然后水平翻转。

代码：

class Solution {

public:

void rotate(vector<vector<int>>& matrix) {

for(int i=0;i<matrix.size();i++)

{

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

{

swap(matrix[i][j],matrix[j][i]);

}

}

for(int i=0;i<matrix.size();i++)

{

for(int j=0;j<matrix.size()/2;j++)

{

swap(matrix[i][j],matrix[i][matrix.size()-j-1]);

}

}

}

};

# 字母异位词分组

思路：

两个字符串互为字母异位词，当且仅当两个字符串包含的字母相同。同一组字母异位词中的字符串具备相同点，可以使用相同点作为一-组字母异位词的标志，使用哈希表存储每一-组字母异位词， 哈希表的键为一组字母异位词的标志，哈希表的值为一组字母异位词列表。

遍历每个字符串，对于每个字符串，得到该字符串所在的一-组字母异位词的标志，将当前字符串加入该组字母异位词的列表中。遍历全部字符串之后，哈希表中的每个键值对即为一-组字母异位词。

方法一:排序

由于互为字母异位词的两个字符串包含的字母相同，因此对两个字符串分别进行排序之后得到的字符串一定是相同的，故可以将排序之后的字符串作为哈希表的键。

代码：

class Solution {

public:

vector<vector<string>> groupAnagrams(vector<string>& strs) {

unordered\_map<string,vector<string>> res;

for(auto& str:strs)

{

auto key=str;

sort(key.begin(),key.end());

res[key].emplace\_back(str);

}

vector<vector<string>> ans;

for(auto it=res.begin();it!=res.end();++it)

{

ans.emplace\_back(it->second);

}

return ans;

}

};

1. 最大子数组和

代码：

class Solution {

public:

int maxSubArray(vector<int>& nums) {

vector<int> dp(nums.size());

if(nums.size()==0) return -1;

if(nums.size()==1) return nums[0];

int i=0;

dp[0]=nums[0];

for(i=1;i<nums.size();i++)

{

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

}

return \*max\_element(dp.begin(),dp.end());

}

};

# 跳跃游戏

思路：

这样以来，我们依次遍历数组中的每一 一个位置，并实时维护最远可以到达的位置。对于当前遍历到的位置x,如果它在最远可以到达的位置的范围内，那么我们就可以从起点通过若干次跳跃到达该位置，因此我们可以用x + nums[x]更新最远可以到达的位置。

在遍历的过程中，如果最远可以到达的位置大于等于数组中的最后-一个位置，那就说明最后-个位置可达，我们就可以直接返回True 作为答案。反之，如果在遍历结束后，最后- -个位置仍然不可达,我们就返回False 作为答案。

我们再来看看题目的示例

[3，2，1，0，4]

●我们一开始在位置0，可以跳跃的最大长度为3,因此最远可以到达的位置被更新为3;

●我们遍历到位置1,由于1≤3，因此位置1可达，加上它可以跳跃的最大长度2得到3,没有超过

最远可以到达的位置;

●位置2、位置3同理，最远可以到达的位置不会被更新;

●我们遍历到位置4,由于4> 3,因此位置4不可达,我们也就不考虑它可以跳跃的最大长度了。

在遍历完成之后，位置4仍然不可达，因此我们返回False 。

代码：

class Solution {

public:

bool canJump(vector<int>& nums) {

int maxpath=0;

for(int i=0;i<nums.size();++i)

{

if(maxpath>=i)

{

maxpath=max(maxpath,i+nums[i]);

}

else

{

return false;

}

}

return true;

}

};

# 合并区间

[合并区间 | 图解排序 + 双指针算法 | 代码清晰易懂 【C++/Java版本】 - 合并区间 - 力扣（LeetCode）](https://leetcode.cn/problems/merge-intervals/solution/by-lin-shen-shi-jian-lu-k-wri9/)

代码：

class Solution {

public:

vector<vector<int>> merge(vector<vector<int>>& intervals) {

sort(intervals.begin(),intervals.end());

vector<vector<int>> ans;

int start=intervals[0][0],end=intervals[0][1];

for(int i=1;i<intervals.size();++i)

{

if(intervals[i][0]>end)

{

ans.push\_back({start,end});

start=intervals[i][0];

end=intervals[i][1];

}

else

{

end=max(intervals[i][1],end);

}

}

ans.push\_back({start,end});

return ans;

}

};

# 不同路径

代码：

class Solution {

public:

int uniquePaths(int m, int n) {

vector<vector<int>> dp(m,vector<int>(n,1));

for(int i=1;i<m;i++)

{

for(int j=1;j<n;j++)

{

dp[i][j]=dp[i-1][j]+dp[i][j-1];

}

}

return dp[m-1][n-1];

}

};

# 最小路径之和

代码：

class Solution {

public:

int minPathSum(vector<vector<int>>& grid) {

if(grid.empty()||grid[0].empty()) return 0;

int m=grid.size(),n=grid[0].size();

vector<vector<int>> dp(m,vector<int>(n));

dp[0][0]=grid[0][0];

for(int i=1;i<m;i++)

{

dp[i][0]=dp[i-1][0]+grid[i][0];

}

for(int j=1;j<n;j++)

{

dp[0][j]=dp[0][j-1]+grid[0][j];

}

for(int i=1;i<m;i++)

{

for(int j=1;j<n;j++)

{

dp[i][j]=min(dp[i-1][j],dp[i][j-1])+grid[i][j];

}

}

return dp[m-1][n-1];

}

};

# 爬楼梯

代码：

class Solution {

public:

int climbStairs(int n) {

if(n==0) return -1;

vector<int> dp(n+1);

dp[0]=1,dp[1]=2;

for(int i=2;i<n;i++)

{

dp[i]=dp[i-1]+dp[i-2];

}

return dp[n-1];

}

};

# 编辑距离

方法：动态规划

思路：

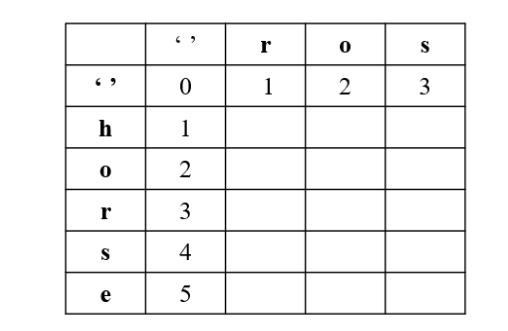
dp[i][j]代表word1到i位置转换成word2到j位置需要最少步数

所以,当word1[i] == word2[j] ，dp[i][j] = dp[i-1][j-1] ;

当word1[i] != word2[j]， dp[i][j] = min(dp[i-1][j-1]，dp[i-1][j]，dp[i][j-1]) + 1

其中，dp[i-1][j-1] 表示替换操作，dp[i-1][j] 表示删除操作，dp[i][j-1] 表示插入操作。

注意，针对第- -行,第一列要单独考虑,我们引入”’下图所示: .



第一行,是word1为空变成word2 最少步数，就是插入操作

第一列，是word2为空，需要的最少步数，就是删除操作

代码：

class Solution {

public:

int minDistance(string word1, string word2) {

if(word1.size()==0&&word2.size()==0) return 0;

word1=' '+word1,word2=' '+word2;

int m=word1.size(),n=word2.size();

vector<vector<int>> dp(m+1,vector<int>(n+1,INT\_MAX));

for(int i=0;i<=m;++i) dp[i][0]=i;

for(int j=0;j<=n;++j) dp[0][j]=j;

for(int i=1;i<=m;++i)

{

for(int j=1;j<=n;++j)

{

if(word1[i]==word2[j]) dp[i][j]=dp[i-1][j-1];

else dp[i][j]=min(min(dp[i-1][j],dp[i][j-1]),dp[i-1][j-1])+1;

}

}

return dp[m][n];

}

};

# 搜索二维矩阵

方法：

找规律：二分法

代码：

class Solution {

public:

bool searchMatrix(vector<vector<int>>& matrix, int target) {

int m=matrix.size(),n=matrix[0].size();

int left=0,right=n\*m-1;

while(left<=right)

{

int mid=(left+right)/2;

int rows=mid/n,columns=mid%n;

if(matrix[rows][columns]==target) return true;

else if(matrix[rows][columns]<target) left=mid+1;

else right=mid-1;

}

return false;

}

};

# 颜色分类

方法一：三指针

代码：

class Solution {

public:

void sortColors(vector<int>& nums) {

int i=0,j=0,k=nums.size()-1;

while(j<=k)

{

if(nums[j]==0) swap(nums[i++],nums[j++]);

else if(nums[j]==1) j++;

else swap(nums[j],nums[k--]);

}

}

};

# 76,。最小覆盖子串

[最小覆盖子串 | 图解滑动窗口 | 最通俗易懂的讲解【c++/java版本】 - 最小覆盖子串 - 力扣（LeetCode）](https://leetcode.cn/problems/minimum-window-substring/solution/leetcode-76-zui-xiao-fu-gai-zi-chuan-cja-lmqz/)

# 子集

代码：

class Solution {

public:

vector<vector<int>> ans;

vector<int> combine;

void dfs(vector<int>& nums,int idx)

{

ans.push\_back(combine);

for(int i=idx;i<nums.size();i++)

{

combine.push\_back(nums[i]);

dfs(nums,i+1);

combine.pop\_back();

}

}

vector<vector<int>> subsets(vector<int>& nums) {

dfs(nums,0);

return ans;

}

};

# 柱状图中最大矩形

思路：

看官方题解

[【柱状图中最大的矩形】单调栈入门，使用单调栈快速寻找边界 - 柱状图中最大的矩形 - 力扣（LeetCode）](https://leetcode.cn/problems/largest-rectangle-in-histogram/solution/84-by-ikaruga/)练习

代码：

class Solution {

public:

//单调栈--单调递增

int largestRectangleArea(vector<int>& heights) {

stack<int> stk;

int ans=0;

heights.push\_back(-1);//遍历结束，将栈清空

for(int i=0;i<heights.size();i++)

{

while(!stk.empty()&&heights[i]<heights[stk.top()])

//如果下一个元素小于栈顶元素，则下一个元素的下标为当前元素最大面积的最右值，

//当前元素最大面积的最左值为小于当前元素的元素的下标，（栈为空为-1）

{

int idx=stk.top();

stk.pop();

int left=stk.empty()?-1:stk.top();

ans=max(ans,(i-left-1)\*heights[idx]);

}

stk.push(i);

}

return ans;

}

};

# 最大矩形

1. 二叉树的中序遍历

方法：迭代（多理解就会了），递归（一般不考）

代码：

class Solution {

public:

vector<int> inorderTraversal(TreeNode\* root) {

vector<int> ans;

stack<TreeNode\*> res;

TreeNode\*cur=root;

while(cur||!res.empty())

{

while(cur)

{

res.push(cur);

cur=cur->left;

}

cur=res.top();

res.pop();

ans.push\_back(cur->val);

cur=cur->right;

}

return ans;

}

};

# 合并两个有序数组

逆序遍历数组：

设置两个指针分别指向两个数组的最后一位，然后遍历两数组，比较两指针指向的元素，

大的放在nums1的最后位置，循环下去；

class Solution {

public:

void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {

int i=m-1,j=n-1,k=m+n-1;

while(i>=0||j>=0)

{

if(i==-1)

{

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

}

else if(j==-1)

{

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

}

else if(nums1[i]>nums2[j])

{

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

}

else

{

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

}

--k;

}

}

};

# 单词搜索

思路：

[单词搜索 | 回溯算法 | 最通俗易懂的讲解 【c++/java版】 - 单词搜索 - 力扣（LeetCode）](https://leetcode.cn/problems/word-search/solution/hui-su-suan-fa-zui-tong-su-yi-dong-de-ji-h2ny/)

代码：

class Solution {

public:

int dx[4]={1,0,-1,0},dy[4]={0,1,0,-1};

bool dfs(vector<vector<char>>& board,string& word,int idx,int x,int y)

{

if(word[idx]!=board[x][y]) return false;

if(idx==word.size()-1) return true;

char t=board[x][y];

board[x][y]='.';

for(int i=0;i<4;i++)

{

int newx=x+dx[i],newy=y+dy[i];

if(newx<0||newx>=board.size()||newy<0||newy>=board[0].size()||board[newx][newy]=='.') continue;

if(dfs(board,word,idx+1,newx,newy)) return true;

}

board[x][y]=t;

return false;

}

bool exist(vector<vector<char>>& board, string word) {

for(int i=0;i<board.size();i++)

{

for(int j=0;j<board[0].size();j++)

{

if(dfs(board,word,0,i,j)) return true;

}

}

return false;

}

};

# 二叉树的中序遍历

一般用迭代法

代码：

class Solution {

public:

vector<int> inorderTraversal(TreeNode\* root) {

vector<int> ans;

TreeNode\*cur=root;

stack<TreeNode\*> stk;

while(cur||!stk.empty())

{

while(cur)

{

stk.push(cur);

cur=cur->left;

}

cur=stk.top();

stk.pop();

ans.push\_back(cur->val);

cur=cur->right;

}

return ans;

}

};

# 不同的二叉搜索树

方法：

动态规划：

●标签:动态规划

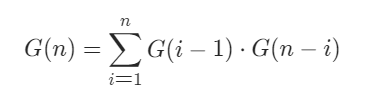
●假设n个节点存在二叉排序树的个数是G(n),令f()为以i为根的二叉搜索树的个数,则G(n)= f(1)+ f(2)+ f(3)+ f(4)+...+ f(n)

●当i为根节点时，其左子树节点个数为i-1个，右子树节点为n-i, 则

f(i)=G(i-1)\*G(n-i)

●综合两个公式可以得到卡特兰数公式

G(n)=G(0)\*G(n-1)+G(1)\*(n-2)+...+G(n-1)\* G(0)



代码：

class Solution {

public:

int numTrees(int n) {

vector<int> f(n+1);

f[0]=1;

for(int i=1;i<=n;i++)

{

for(int j=1;j<=i;j++)

{

f[i]+=f[j-1]\*f[i-j];

}

}

return f[n];

}

};

# 验证二叉搜索树

方法：二叉树的中序遍历

代码：

/\*\*

\* Definition for a binary tree node.

\* struct TreeNode {

\* int val;

\* TreeNode \*left;

\* TreeNode \*right;

\* TreeNode() : val(0), left(nullptr), right(nullptr) {}

\* TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}

\* TreeNode(int x, TreeNode \*left, TreeNode \*right) : val(x), left(left), right(right) {}

\* };

\*/

class Solution {

public:

bool isValidBST(TreeNode\* root) {

long long pre=(long long)INT\_MIN-1;

stack<TreeNode\*> stk;

TreeNode\*cur=root;

while(cur||!stk.empty())

{

while(cur)

{

stk.push(cur);

cur=cur->left;

}

cur=stk.top();

stk.pop();

if(cur->val<=pre) return false;

pre=cur->val;

cur=cur->right;

}

return true;

}

};

# 对称二叉树

方法：迭代队列

代码：

class Solution {

public:

bool check(TreeNode\*p,TreeNode\*q)

{

queue<TreeNode\*> ans;//用队列保存结点

ans.push(p);ans.push(q);//将跟结点的左右结点放入队列

while(!ans.empty())

{

p=ans.front(); ans.pop();//从队列中取出两节点，比较两节点

q=ans.front(); ans.pop();

if(!p&&!q) continue;//如果两节点为空，就继续循环。

if((!p||!q)||(p->val!=q->val)) return false;

ans.push(p->left);

ans.push(q->right);

ans.push(p->right);

ans.push(q->left);

}

return true;

}

bool isSymmetric(TreeNode\* root) {

return check(root,root);

}

};

# 二叉树的层序遍历

步骤：

1. 先定义一个结果数组，如果根节点为空，返回空结果队列。
2. 在定义一个队列，将根节点插入队列。
3. 队列不为空，循环。计数队列的元素，定义一个容器，存放当前一层的元素的值。
4. 根据计数，保存弹出队列的元素，并把该元素放在容器中，再把该元素的左右孩子放入队列。
5. 循环结束，将容器放入结果数组中。

代码：

class Solution {

public:

vector<vector<int>> levelOrder(TreeNode\* root) {

vector<vector<int>> ans;/\* 根据函数返回值定义存储结果的变量 \*/

if(!root) return ans;/\* 判空 \*/

queue<TreeNode\*> q;/\* 定义一个队列用于存储节点的数据 \*/

q.push(root);

while(!q.empty())/\* 开始层序遍历 \*/

{

int cnt=q.size();/\* 计算队列的大小也即有多少个孩子 \*/

vector<int> cur;/\* 定义一个临时vector 存储每一层 \*/

while(cnt--)

{

TreeNode\*node=q.front();/\* 获取第一个节点数据 \*/

q.pop();

cur.push\_back(node->val);

if(node->left) q.push(node->left);

if(node->right) q.push(node->right);

}

ans.push\_back(cur);/\* 将一层的数据保存 \*/

}

return ans;

}

};

# 二叉树的最大深度

方法一：层序遍历，便利一层，计数+1；

代码：

class Solution {

public:

int maxDepth(TreeNode\* root) {

if(!root) return 0;

queue<TreeNode\*> q;

q.push(root);

int ans=0;

while(!q.empty())

{

int cnt=q.size();

while(cnt--)

{

TreeNode\*node=q.front();

q.pop();

if(node->left) q.push(node->left);

if(node->right) q.push(node->right);

}

ans++;

}

return ans;

}

};

方法二：递归

代码：

class Solution {

public:

int maxDepth(TreeNode\* root) {

if(!root) return 0;

return max(maxDepth(root->left),maxDepth(root->right))+1;

}

};

# [105 从前序与中序遍历序列构造二叉树](https://leetcode.cn/problems/construct-binary-tree-from-preorder-and-inorder-traversal/)

方法：递归

代码：

class Solution {

public:

unordered\_map<int,int> pos;

TreeNode\* buildTree(vector<int>& preorder, vector<int>& inorder) {

int n=preorder.size();

for(int i=0;i<n;i++) pos[inorder[i]]=i;

return build(preorder,inorder,0,n-1,0,n-1);

}

TreeNode\*build(vector<int>& preorder,vector<int>& inorder,int pl,int pr,int il,int ir)

{

if(pl>pr||il>ir) return nullptr;

int k=pos[preorder[pl]]-il;

TreeNode\*root=new TreeNode(preorder[pl]);

root->left=build(preorder,inorder,pl+1,pl+k,il,il+k-1);

root->right=build(preorder,inorder,pl+k+1,pr,il+k+1,ir);

return root;

}

};

# 二叉树展开为链表

方法：

1.将原来的右子树接到左子树的最右边节点

2.将左子树插入到右子树的地方

3.考虑新的右子树的根节点，- -直重复上边的过程,直到新的右子树为null

代码：

class Solution {

public:

void flatten(TreeNode\* root) {

TreeNode\*cur=root;

while(cur)

{

if(cur->left)

{

TreeNode\*p=cur->left;

while(p->right) p=p->right;

p->right=cur->right;

cur->right=cur->left;

cur->left=nullptr;

}

cur=cur->right;

}

}

};

# 买卖股票的最佳时机

代码：

class Solution {

public:

int maxProfit(vector<int>& prices) {

int ans=0,minprice=prices[0];

for(int i=1;i<prices.size();i++)

{

ans=max(ans,prices[i]-minprice);

minprice=min(minprice,prices[i]);

}

return ans;

}

};

# 二叉树的最大路径和

[二叉树中的最大路径和 | 图解递归做法 | 代码简洁易懂 【C++/Java版本】 - 二叉树中的最大路径和 - 力扣（LeetCode）](https://leetcode.cn/problems/binary-tree-maximum-path-sum/solution/by-lin-shen-shi-jian-lu-k-2td7/)

代码：

class Solution {

public:

int res = INT\_MIN;

int maxPathSum(TreeNode\* root) {

dfs(root);

return res;

}

int dfs(TreeNode\* root){

if(!root) return 0;

int left = max(0, dfs(root->left)), right = max(0, dfs(root->right));

res = max(res, root->val + left + right);

return root->val + max(left, right);

}

};

# 最长连续子序列

[最长连续序列 | 哈希 | 代码简洁易懂 【c++/java版】 - 最长连续序列 - 力扣（LeetCode）](https://leetcode.cn/problems/longest-consecutive-sequence/solution/ha-xi-zui-qing-xi-yi-dong-de-jiang-jie-c-xpnr/)

方法：哈希表

代码：

class Solution {

public:

int longestConsecutive(vector<int>& nums) {

unordered\_set<int> st(nums.begin(),nums.end());

int ans=0;

for(auto& num:st)

{

if(!st.count(num-1))

{

int x=num;

while(st.count(x)) x++;

ans=max(ans,x-num);

}

}

return ans;

}

};

# 只出现一次的数字

方法：异或运算

代码：

class Solution {

public:

int singleNumber(vector<int>& nums) {

int ans=0;

for(auto& num:nums)

{

ans^=num;

}

return ans;

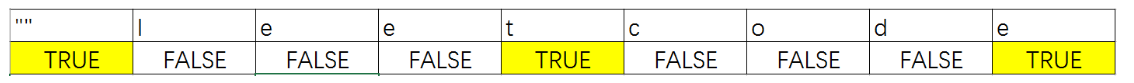
}

};

# 单词拆分

方法：动态规划

[单词拆分 - 单词拆分 - 力扣（LeetCode）](https://leetcode.cn/problems/word-break/solution/dan-ci-chai-fen-by-leetcode-solution/)



代码：

class Solution {

public:

bool wordBreak(string s, vector<string>& wordDict) {

unordered\_set<string> st(wordDict.begin(),wordDict.end());

int n=s.length();

s=' '+s;

vector<bool> dp(n+1);

dp[0]=true;

for(int i=1;i<=n;i++)

{

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

{

if(dp[j]&&st.count(s.substr(j+1,i-j)))

{

dp[i]=true;

break;

}

}

}

return dp[n];

}

};

# 环形链表

方法一:哈希表

思路及算法

最容易想到的方法是遍历所有节点，每次遍历到一个节点时,判断该节点此前是否被访问过。

具体地，我们可以使用哈希表来存储所有已经访问过的节点。每次我们到达-一个节点，如果该节点已经存在于哈希表中，则说明该链表是环形链表，否则就将该节点加入哈希表中。重复这一过程，直到我们遍历

完整个链表即可。

代码：

class Solution {

public:

bool hasCycle(ListNode \*head) {

unordered\_set<ListNode\*> ans;

while(head)

{

if(ans.count(head)) return true;

ans.insert(head);

head=head->next;

}

return false;

}

};

方法二：

我们可以根据上述思路来解决本题。具体地，我们定义两个指针，- -快一满。慢指针每次只移动一-步,而快指针每次移动两步。初始时，慢指针在位置head ，而快指针在位置head. next。这样- -来， 如果在移动的过程中，快指针反过来追上慢指针，就说明该链表为环形链表。否则快指针将到达表尾部，该链表不为环形链表。

代码：

class Solution {

public:

bool hasCycle(ListNode \*head) {

if(!head||!head->next) return false;

ListNode\*slow=head;

ListNode\*fast=head->next;

while(slow!=fast)

{

if(!fast||!fast->next) return false;

slow=slow->next;

fast=fast->next->next;

}

return true;

}

};

1. 环形链表2

代码：

class Solution {

public:

ListNode \*detectCycle(ListNode \*head) {

ListNode\*fast=head,\*slow=head;

do

{

if(!fast||!fast->next) return NULL;

fast=fast->next->next;

slow=slow->next;

}while(slow!=fast);

fast=head;

while(slow!=fast)

{

fast=fast->next;

slow=slow->next;

}

return fast;

}

};

# LRU缓冲（重点）

代码：

struct DLinkedNode

{

int key,value;

DLinkedNode\*prev;

DLinkedNode\*next;

DLinkedNode():key(0),value(0),prev(nullptr),next(nullptr){}

DLinkedNode(int \_key,int \_value):key(\_key),value(\_value),prev(nullptr),next(nullptr){}

};

class LRUCache {

private:

unordered\_map<int,DLinkedNode\*> cache;

DLinkedNode\*head;

DLinkedNode\*tail;

int size;

int capacity;

public:

LRUCache(int \_capacity):capacity(\_capacity),size(0) {

head=new DLinkedNode();

tail=new DLinkedNode();

head->next=tail;

tail->prev=head;

}

int get(int key) {

if(!cache.count(key)) return -1;

DLinkedNode\*node=cache[key];

moveToHead(node);

return node->value;

}

void put(int key, int value) {

if(!cache.count(key))

{

DLinkedNode\*node=new DLinkedNode(key,value);

cache[key]=node;

addToHead(node);

++size;

if(size>capacity)

{

DLinkedNode\*removed=removeTail();

cache.erase(removed->key);

delete removed;

--size;

}

}

else

{

DLinkedNode\*node=cache[key];

node->value=value;

moveToHead(node);

}

}

void addToHead(DLinkedNode\*node)

{

node->prev=head;

node->next=head->next;

head->next->prev=node;

head->next=node;

}

void removeNode(DLinkedNode\*node)

{

node->prev->next=node->next;

node->next->prev=node->prev;

}

void moveToHead(DLinkedNode\*node)

{

removeNode(node);

addToHead(node);

}

DLinkedNode\*removeTail()

{

DLinkedNode\*node=tail->prev;

removeNode(node);

return node;

}

};

# 对链表进行插入排序

思路：

[对链表进行插入排序 - 对链表进行插入排序 - 力扣（LeetCode）](https://leetcode.cn/problems/insertion-sort-list/solution/dui-lian-biao-jin-xing-cha-ru-pai-xu-by-leetcode-s/)

代码：

class Solution {

public:

ListNode\* insertionSortList(ListNode\* head) {

if(head==nullptr) return head;

ListNode\*dummy=new ListNode(0);

dummy->next=head;

ListNode\*lastsorted=head;

ListNode\*cur=head->next;

while(cur)

{

if(lastsorted->val<=cur->val)

{

lastsorted=lastsorted->next;

}

else

{

ListNode\*prev=dummy;

while(prev->next->val<=cur->val)

{

prev=prev->next;

}

lastsorted->next=cur->next;

cur->next=prev->next;

prev->next=cur;

}

cur=lastsorted->next;

}

return dummy->next;

}

};

# 排序链表。

思路。看看代码，理解。

[排序链表 - 排序链表 - 力扣（LeetCode）](https://leetcode.cn/problems/sort-list/solution/pai-xu-lian-biao-by-leetcode-solution/)

1. 乘积最大子数组

思路：

[乘积最大子数组 - 乘积最大子数组 - 力扣（LeetCode）](https://leetcode.cn/problems/maximum-product-subarray/solution/cheng-ji-zui-da-zi-shu-zu-by-leetcode-solution/)

代码：

class Solution {

public:

int maxProduct(vector<int>& nums) {

vector<int> maxF(nums),minF(nums);

for(int i=1;i<nums.size();i++)

{

maxF[i]=max(maxF[i-1]\*nums[i],max(nums[i],minF[i-1]\*nums[i]));

minF[i]=min(maxF[i-1]\*nums[i],min(nums[i],minF[i-1]\*nums[i]));

}

return \*max\_element(maxF.begin(),maxF.end());

}

};

# 最小栈

思路：

设置两个栈，一个保存元素，一个保存最小值。

我们只需要设计-一个数据结构,使得每个元素a与其相应的最小值m时刻保持

对应。因此我们可以使用一个辅助栈，与元素栈同步插入与删除，用于存储与每个元素对应的最小值。

●当一个元素要入栈时，我们取当前辅助栈的栈顶存储的最小值，与当前元素比较得出最小值，将这

个最小值插入辅助栈中;

●当一个元素要出栈时，我们把辅助栈的栈顶元素也一并弹出;

●在任意-一个时刻，栈内元素的最小值就存储在辅助栈的栈顶元素中。

代码：

class MinStack {

public:

stack<int> data\_stack;

stack<int> min\_stack;

MinStack() {

min\_stack.push(INT\_MAX);

}

void push(int val) {

data\_stack.push(val);

min\_stack.push(min(min\_stack.top(),val));

}

void pop() {

data\_stack.pop();

min\_stack.pop();

}

int top() {

return data\_stack.top();

}

int getMin() {

return min\_stack.top();

}

};

/\*\*

\* Your MinStack object will be instantiated and called as such:

\* MinStack\* obj = new MinStack();

\* obj->push(val);

\* obj->pop();

\* int param\_3 = obj->top();

\* int param\_4 = obj->getMin();

\*/

# 相交链表

方法：

一种比较巧妙的方式是，分别为链表A和链表B设置指针A和指针B，然后开始遍历链表，如果遍历完当前链表，则将指针指向另外一个链表的头部继续遍历，直至两个指针相遇。

最终两个指针分别走过的路径为：

指针A :a+c+b

指针B :b+c+a

明显 a+c+b = b+c+a,因而如果两个链表相交，则指针A和指针B必定在相交结点相遇。

代码：

class Solution {

public:

ListNode \*getIntersectionNode(ListNode \*headA, ListNode \*headB) {

ListNode \*A = headA, \*B = headB;

while (A != B) {

A = A != nullptr ? A->next : headB;

B = B != nullptr ? B->next : headA;

}

return A;

}

};

# 多数元素

方法一：哈希表法，{key,value},key为元素，value为元素出现的次数

代码；class Solution {

public:

int majorityElement(vector<int>& nums) {

unordered\_map<int,int> counts;

for(auto &it:nums)

{

++counts[it];

}

int ans=0,res=0;

for(auto &it:counts)

{

if(it.second>res)

{

res=it.second;

ans=it.first;

}

}

return ans;

}

};

方法二：排序

如果将数组nums中的所有元素按照单调递增或单调递减的顺序排序，那么下标为n/2

的元素(下标从0开始)一定是众数。

代码：class Solution {

public:

int majorityElement(vector<int>& nums) {

sort(nums.begin(), nums.end());

return nums[nums.size() / 2];

}

};

方法三：

”同归于尽消杀法”:

于多数超过50%,比如100个数，那么多数至少51个,剩下少数是49个。

1.遍历数组

2.第一个到来的士兵，直接插上自己阵营的旗帜占领这块高地，此时领主winner就是这个阵营的人，现存兵力count= 1。

3.如果新来的士兵和前一个兵是同-阵营,则集合起来占领高地，领主不变, winner 依然是当前这个士兵所属阵营，现存兵幼count

加一;

4.如果新来到的兵不同一阵营，则前方阵营派一个士兵和它同归于尽。 此时前方阵营兵力-1, 即使双方都死光，这块高地的旗帜

winner不变,没有可以去换上自己的新旗帜。

5.当下一个兵到来,发现前方阵营已经没有兵助,新士兵就成了领主, winner 变成这个兵所属阵营的旗帜，现存兵励count ++。

6.就这样各路军阀一直厮杀以-敌-同归于尽的方式下去，直到少数阵营都死光，剩下几个必然属于多数阵营的，winner 多数阵营。

(多数阵营51个,少数阵营只有49个,死剩下的2个就是多数阵营的人)

代码：

class Solution {

public:

int majorityElement(vector<int>& nums) {

if(nums.size()==0) return -1;

int winner=nums[0],count=1;

for(int i=1;i<nums.size();i++)

{

if(nums[i]==winner) ++count;

else if(count==0)

{

winner=nums[i];

++count;

}

else

{

count--;

}

}

return winner;

}

};

# 打家劫舍

动态规划

代码：

class Solution {

public:

int rob(vector<int>& nums) {

int n=nums.size();

if(n==0) return -1;

if(n==1) return nums[0];

if(n==2) return max(nums[0],nums[1]);

vector<int> dp(n);

dp[0]=nums[0];

dp[1]=max(nums[0],nums[1]);

for(int i=2;i<n;i++)

{

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

}

return dp[n-1];

}

};

# 岛屿数量

方法一：深度优先搜索遍历

代码：class Solution {

private:

void dfs(vector<vector<char>>&grid,int i,int j)

{

int rows=grid.size();

int colums=grid[0].size();

grid[i][j]='0';

if(i-1>=0&&grid[i-1][j]=='1') dfs(grid,i-1,j);

if(i+1<rows&&grid[i+1][j]=='1') dfs(grid,i+1,j);

if(j-1>=0&&grid[i][j-1]=='1') dfs(grid,i,j-1);

if(j+1<colums&&grid[i][j+1]=='1') dfs(grid,i,j+1);

}

public:

int numIslands(vector<vector<char>>& grid) {

int rows=grid.size();

int colums=grid[0].size();

if(!rows||!colums) return 0;

int count=0;

for(int i=0;i<rows;++i)

{

for(int j=0;j<colums;++j)

{

if(grid[i][j]=='1')

{

dfs(grid,i,j);

++count;

}

}

}

return count;

}

};

方法二：广度优先搜索遍历

方法三：并查集

# 反转链表

方法一：迭代法[简单易懂Java/C++ /Python/js/go 动画讲解 - 反转链表 - 反转链表 - 力扣（LeetCode）](https://leetcode.cn/problems/reverse-linked-list/solution/shi-pin-jiang-jie-die-dai-he-di-gui-hen-hswxy/)

代码：class Solution {

public:

ListNode\* reverseList(ListNode\* head) {

if(!head) return head;

ListNode\*prev=NULL;

ListNode\*cur=head;

while(cur)

{

ListNode\*Next=cur->next;

cur->next=prev;

prev=cur;

cur=Next;

}

return prev;

}

};

方法二：递归

代码：class Solution {

public:

ListNode\* reverseList(ListNode\* head) {

if(!head||!head->next) return head;

/\*

直到当前节点的下一个节点为空时返回当前节点

由于5没有下一个节点了，所以此处返回节点5

\*/

ListNode\* p=reverseList(head->next);

//递归传入下一个节点，目的是为了到达最后一个节点

head->next->next=head;

head->next=NULL;

/\*

第一轮出栈，head为5，head.next为空，返回5

第二轮出栈，head为4，head.next为5，执行head.next.next=head也就是5.next=4，

把当前节点的子节点的子节点指向当前节点

此时链表为1->2->3->4<->5，由于4与5互相指向，所以此处要断开4.next=null

此时链表为1->2->3->4<-5

返回节点5

第三轮出栈，head为3，head.next为4，执行head.next.next=head也就是4.next=3，

此时链表为1->2->3<->4<-5，由于3与4互相指向，所以此处要断开3.next=null

此时链表为1->2->3<-4<-5

返回节点5

第四轮出栈，head为2，head.next为3，执行head.next.next=head也就是3.next=2，

此时链表为1->2<->3<-4<-5，由于2与3互相指向，所以此处要断开2.next=null

此时链表为1->2<-3<-4<-5

返回节点5

第五轮出栈，head为1，head.next为2，执行head.next.next=head也就是2.next=1，

此时链表为1<->2<-3<-4<-5，由于1与2互相指向，所以此处要断开1.next=null

此时链表为1<-2<-3<-4<-5

返回节点5

出栈完成，最终头节点5->4->3->2->1

\*/

return p;

}

};

# 课程表

了解了一下，争取会写。

# 208：实现Trie（前缀树）（重点）

代码：class Trie {

private:

bool isEnd;

Trie\*next[26];

public:

Trie() {

isEnd=false;

memset(next,0,sizeof(next));

}

void insert(string word) {

Trie\*node=this;

for(char c:word){

if(node->next[c-'a']==NULL){

node->next[c-'a']=new Trie();

}

node=node->next[c-'a'];

}

node->isEnd=true;

}

bool search(string word) {

Trie\*node=this;

for(char c:word){

node=node->next[c-'a'];

if(node==NULL){

return false;

}

}

return node->isEnd;

}

bool startsWith(string prefix) {

Trie\*node=this;

for(char c:prefix){

node=node->next[c-'a'];

if(node==NULL){

return false;

}

}

return true;

}

};

/\*\*

\* Your Trie object will be instantiated and called as such:

\* Trie\* obj = new Trie();

\* obj->insert(word);

\* bool param\_2 = obj->search(word);

\* bool param\_3 = obj->startsWith(prefix);

\*/

# 数组中第K大个元素

就是排序求值。用堆堆或快排

代码：

class Solution {

public:

void heapify(vector<int>& nums,int n,int i)

{

if(i>n) return ;

int maxNode=i;

int lson=i\*2+1;

int rson=i\*2+2;

if(lson<n&&nums[lson]>nums[maxNode]) maxNode=lson;

if(rson<n&&nums[rson]>nums[maxNode]) maxNode=rson;

if(maxNode!=i)

{

swap(nums[maxNode],nums[i]);

heapify(nums,n,maxNode);

}

}

void build\_sort(vector<int>& nums,int n,int k)

{

for(int i=n/2-1;i>=0;i--)

{

heapify(nums,n,i);

}

for(int i=n-1;i>=n-k;i--)

{

swap(nums[i],nums[0]);

heapify(nums,i,0);

}

}

int findKthLargest(vector<int>& nums, int k) {

if(nums.size()==0) return 0;

int n=nums.size();

build\_sort(nums,n,k);

return nums[n-k];

}

};

# 最大正方形

方法：动态规划

dp[i][j]=min(min(dp[i-1][j],dp[i][j-1]),dp[i-1][j-1])+1;

代码：class Solution {

public:

int maximalSquare(vector<vector<char>>& matrix) {

if(matrix.size()==0||matrix[0].size()==0) return 0;

int maxSide=0;

int rows=matrix.size(),columns=matrix[0].size();

vector<vector<int>> dp(rows,vector<int>(columns));

for(int i=0;i<rows;i++)

{

for(int j=0;j<columns;j++)

{

if(matrix[i][j]=='1')

{

if(i==0||j==0) dp[i][j]=1;

else dp[i][j]=min(min(dp[i-1][j],dp[i][j-1]),dp[i-1][j-1])+1;

}

maxSide=max(maxSide,dp[i][j]);

}

}

return pow(maxSide,2);

}

};

1. 翻转二叉树

代码：class Solution {

public:

void dfs(TreeNode\*root)

{

if(!root) return;

TreeNode\*temp=root->left;

root->left=root->right;

root->right=temp;

dfs(root->left);

dfs(root->right);

}

TreeNode\* invertTree(TreeNode\* root) {

dfs(root);

return root;

}

};

# 回文链表

方法：

整个流程可以分为以下五个步骤:

1.找到前半部分链表的尾节点。

2.反转后半部分链表。

3.判断是否回文。

4.恢复链表。

5.返回结果。

代码：class Solution {

public:

bool isPalindrome(ListNode\* head) {

if(!head) return true;

ListNode\*middlenode=middleNode(head);

ListNode\*mid= reverseList(middlenode->next);

while(mid)

{

if(head->val!=mid->val) return false;

head=head->next;

mid=mid->next;

}

return true;

}

ListNode\* reverseList(ListNode\*node)

{

ListNode\*prev=nullptr;

ListNode\*cur=node;

while(cur)

{

ListNode\*Next=cur->next;

cur->next=prev;

prev=cur;

cur=Next;

}

return prev;

}

ListNode\* middleNode(ListNode\*node)

{

ListNode\*fast=node;

ListNode\*slow=node;

while(fast->next&&fast->next->next)

{

fast=fast->next->next;

slow=slow->next;

}

return slow;

}

};

# 二叉树的最近公共祖先

代码：

[236. 二叉树的最近公共祖先（DFS ，清晰图解） - 二叉树的最近公共祖先 - 力扣（LeetCode）](https://leetcode.cn/problems/lowest-common-ancestor-of-a-binary-tree/solution/236-er-cha-shu-de-zui-jin-gong-gong-zu-xian-hou-xu/)

class Solution {

public:

TreeNode\* lowestCommonAncestor(TreeNode\* root, TreeNode\* p, TreeNode\* q) {

if(!root||root==p||root==q) return root;

TreeNode\*left=lowestCommonAncestor(root->left,p,q);

TreeNode\*right=lowestCommonAncestor(root->right,p,q);

if(!left) return right;

if(!right) return left;

if(!left&&!right) return nullptr;

return root;

}

};

# 除自身以外数组的乘积

代码：

class Solution {

public:

vector<int> productExceptSelf(vector<int>& nums) {

int flag=1;

vector<int> front(nums.size());

for(int i=0;i<nums.size();i++)

{

front[i]=flag;

flag\*=nums[i];

}

vector<int> back(nums.size());

flag=1;

for(int i=nums.size()-1;i>=0;i--)

{

back[i]=flag;

flag\*=nums[i];

}

for(int i=0;i<nums.size();i++)

{

nums[i]=front[i]\*back[i];

}

return nums;

}

};

优化代码：

class Solution {

public:

vector<int> productExceptSelf(vector<int>& nums) {

int n=nums.size();

vector<int> answer(n);

answer[0]=1;

for(int i=1;i<n;i++)

{

answer[i]=nums[i-1]\*answer[i-1];

}

int flag=1;

for(int i=n-1;i>=0;i--)

{

answer[i]=answer[i]\*flag;

flag\*=nums[i];

}

return answer;

}

};

# 滑动窗口最大值

方法一：优先队列

[滑动窗口最大值 - 滑动窗口最大值 - 力扣（LeetCode）](https://leetcode.cn/problems/sliding-window-maximum/solution/hua-dong-chuang-kou-zui-da-zhi-by-leetco-ki6m/)

代码：

class Solution {

public:

vector<int> maxSlidingWindow(vector<int>& nums, int k) {

int n = nums.size();

priority\_queue<pair<int, int>> q;

for (int i = 0; i < k; ++i) {

q.emplace(nums[i], i);

}

vector<int> ans = {q.top().first};

for (int i = k; i < n; ++i) {

q.emplace(nums[i], i);

while (q.top().second <= i - k) {

q.pop();

}

ans.push\_back(q.top().first);

}

return ans;

}

};

方法二：单调队列（双端队列deque）

[动画演示 单调队列 239.滑动窗口最大值 - 滑动窗口最大值 - 力扣（LeetCode）](https://leetcode.cn/problems/sliding-window-maximum/solution/dong-hua-yan-shi-dan-diao-dui-lie-239hua-hc5u/)

代码：

class Solution {

public:

vector<int> maxSlidingWindow(vector<int>& nums, int k) {

int n = nums.size();

deque<int> q;

for (int i = 0; i < k; ++i) {

while (!q.empty() && nums[i] >= nums[q.back()]) {

q.pop\_back();

}

q.push\_back(i);

}

vector<int> ans = {nums[q.front()]};

for (int i = k; i < n; ++i) {

while (!q.empty() && nums[i] >= nums[q.back()]) {

q.pop\_back();

}

q.push\_back(i);

while (q.front() <= i - k) {

q.pop\_front();

}

ans.push\_back(nums[q.front()]);

}

return ans;

}

};

# 二维矩阵II

1. 方法：找规律

代码：

class Solution {

public:

bool searchMatrix(vector<vector<int>>& matrix, int target) {

int columns=matrix[0].size()-1,rows=0;

while(columns>=0&&rows<=matrix.size()-1)

{

if(matrix[rows][columns]==target) return true;

else if(matrix[rows][columns]<target) ++rows;

else columns--;

}

return false;

}

};

方法二：二分法

代码：

class Solution {

public:

bool searchMatrix(vector<vector<int>>& matrix, int target) {

for(auto&rows:matrix)

{

auto it=lower\_bound(rows.begin(),rows.end(),target);

if(it!=rows.end()&&target==\*it) return true;

}

return false;

}

};

# 完全平方数

[画解算法：279. 完全平方数 - 完全平方数 - 力扣（LeetCode）](https://leetcode.cn/problems/perfect-squares/solution/hua-jie-suan-fa-279-wan-quan-ping-fang-shu-by-guan/)

代码：

class Solution {

public:

int numSquares(int n) {

vector<int> dp(n+1,0);

if(n==0) return 0;

for(int i=1;i<n+1;i++)

{

dp[i]=i;

for(int j=1;j\*j<=i;j++)

{

dp[i]=min(dp[i-j\*j]+1,dp[i]);

}

}

return dp[n];

}

};

# 移动零

方法：双指针

思路及解法.

使用双指针，左指针指向当前已经处理好的序列的尾部，右指针指向待处理序列的头部。

右指针不断向右移动，每次右指针指向非零数,则将左右指针对应的数交换，同时左指针右移。

注意到以下性质:

1.左指针左边均为非零数;

2.右指针左边直到左指针处均为零。

因此每次交换，都是将左指针的零与右指针的非零数交换，且非零数的相对顺序并未改变。

代码：

class Solution {

public:

void moveZeroes(vector<int>& nums) {

int n=nums.size(),left=0,right=0;

while(right<n)

{

if(nums[right])

{

swap(nums[left],nums[right]);

left++;

}

right++;

}

}

};

方法二：

代码：

class Solution {

public:

void moveZeroes(vector<int>& nums) {

int index=0,n=nums.size();

for(auto num:nums)

{

if(num!=0) nums[index++]=num;

}

for(int i=index;i<n;i++)

{

nums[i]=0;

}

}

};

# 寻找重复数

方法：

[287.寻找重复数 - 寻找重复数 - 力扣（LeetCode）](https://leetcode.cn/problems/find-the-duplicate-number/solution/287xun-zhao-zhong-fu-shu-by-kirsche/)

代码：

class Solution {

public:

int findDuplicate(vector<int>& nums) {

int slow = 0, fast = 0;

do {

slow = nums[slow];

fast = nums[nums[fast]];

} while (slow != fast);

slow = 0;

while (slow != fast) {

slow = nums[slow];

fast = nums[fast];

}

return slow;

}

};

# 最长递增子序列

方法：动态规划

代码：class Solution {

public:

int lengthOfLIS(vector<int>& nums) {

int n=nums.size();

if(n==0) return 0;

vector<int> dp(n,0);

for(int i=0;i<n;++i)

{

dp[i]=1;

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

{

if(nums[i]>nums[j]){

dp[i]=max(dp[i],dp[j]+1);

}

}

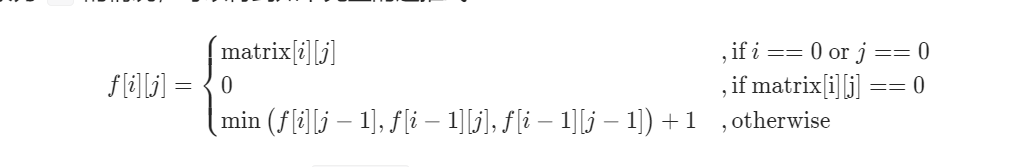
}

return \*max\_element(dp.begin(),dp.end());

}

};

# 统计全为1的正方形子矩阵



代码：class Solution {

public:

int countSquares(vector<vector<int>>& matrix) {

if(matrix.size()==0||matrix[0].size()==0) return 0;

int maxNum=0;

int rows=matrix.size(),columns=matrix[0].size();

vector<vector<int>> dp(rows,vector<int>(columns));

for(int i=0;i<rows;i++)

{

for(int j=0;j<columns;j++)

{

if(i==0||j==0) dp[i][j]=matrix[i][j];

else if(matrix[i][j]==0) dp[i][j]=0;

else dp[i][j]=min(min(dp[i-1][j],dp[i][j-1]),dp[i-1][j-1])+1;

maxNum+=dp[i][j];

}

}

return maxNum;

}

};

# 最佳买卖股票含最佳冷冻期

方法：[最佳买卖股票时机含冷冻期 - 最佳买卖股票时机含冷冻期 - 力扣（LeetCode）](https://leetcode.cn/problems/best-time-to-buy-and-sell-stock-with-cooldown/solution/zui-jia-mai-mai-gu-piao-shi-ji-han-leng-dong-qi-4/)

代码：

class Solution {

public:

int maxProfit(vector<int>& prices) {

if(prices.empty()) return 0;

int n=prices.size();

vector<vector<int>> f(n,vector<int>(3));

f[0][0]=-prices[0];

for(int i=1;i<n;++i)

{

f[i][0]=max(f[i-1][0],f[i-1][2]-prices[i]);

f[i][1]=f[i-1][0]+prices[i];

f[i][2]=max(f[i-1][1],f[i-1][2]);

}

return max(f[n-1][1],f[n-1][2]);

}

};

# 零钱兑换（腾讯二面）（重点没理解了，背包问题）

方法：[零钱兑换 - 零钱兑换 - 力扣（LeetCode）](https://leetcode.cn/problems/coin-change/solution/322-ling-qian-dui-huan-by-leetcode-solution/)

代码；

class Solution {

public:

int coinChange(vector<int>& coins, int amount) {

if(coins.size()==0) return -1;

int Max=amount+1;

vector<int> dp(amount+1,Max);

dp[0]=0;

for(int i=1;i<=amount;++i)

{

for(int j=0;j<coins.size();++j)

{

if(coins[j]<=i)

{

dp[i]=min(dp[i],dp[i-coins[j]]+1);

}

}

}

return dp[amount]>amount?-1:dp[amount];

}

};

# 打家劫舍

方法：动态规划

代码：

class Solution {

public:

unordered\_map<TreeNode\*,int>f,g;

void dfs(TreeNode\*node)

{

if(!node) return ;

dfs(node->left);

dfs(node->right);

f[node]=node->val+g[node->left]+g[node->right];

g[node]=max(f[node->left],g[node->left])+max(f[node->right],g[node->right]);

}

int rob(TreeNode\* root) {

dfs(root);

return max(f[root],g[root]);

}

};

# 比特位计算1

方法一：奇偶数验证--[清晰的思路 - 比特位计数 - 力扣（LeetCode）](https://leetcode.cn/problems/counting-bits/solution/hen-qing-xi-de-si-lu-by-duadua/)

代码：class Solution {

public:

vector<int> countBits(int n) {

vector<int> ans(n+1);

ans[0]=0;

for(int i=1;i<=n;i++)

{

if(i%2==1){

ans[i]=ans[i-1]+1;

}else{

ans[i]=ans[i/2];

}

}

return ans;

}

};

方法二：位与

代码：class Solution {

public:

int countOnes(int x) {

int ones = 0;

while (x > 0) {

x &= (x - 1);

ones++;

}

return ones;

}

vector<int> countBits(int n) {

vector<int> bits(n + 1);

for (int i = 0; i <= n; i++) {

bits[i] = countOnes(i);

}

return bits;

}

};

# 求两个相同字符之间的最长字符串。

思路：建立一个哈希表m，变量ans=-1;遍历给定字符串，从哈希表中找字母，不存在就将其插入，value为i，否者就ans=max(ans,i-m[s[i]-1);

class Solution {

public:

int maxLengthBetweenEqualCharacters(string s) {

unordered\_map<char,int> m;

if(s.size()==0) return -1;

int ans=-1;

for(int i=0;i<s.size();++i)

{

if(m.find(s[i])==m.end()) m[s[i]]=i;

else ans=max(ans,i-m[s[i]]-1);

}

return ans;

}

};