CONTENT

[TopCode 1](#_Toc142841209)

[无重复字符的最长子串 1](#_Toc142841210)

[反转链表 1](#_Toc142841211)

[LRU 缓存 2](#_Toc142841212)

[数组中的第k大的元素 3](#_Toc142841213)

[K个一组翻转链表 4](#_Toc142841214)

[三数之和 4](#_Toc142841215)

[最大子数组和 5](#_Toc142841216)

[快速排序 5](#_Toc142841217)

[合并两个有序链表 6](#_Toc142841218)

[合并K个有序链表 6](#_Toc142841219)

[两数之和 7](#_Toc142841220)

[二叉树的层序遍历 7](#_Toc142841221)

[最长回文子串 8](#_Toc142841222)

[搜索旋转排序数组 8](#_Toc142841223)

[环形链表 9](#_Toc142841224)

[买卖股票的最佳时机 9](#_Toc142841225)

[合并两个有序数组 10](#_Toc142841226)

[岛屿数量 10](#_Toc142841227)

[二叉树的最近公共祖先 11](#_Toc142841228)

[二叉树的锯齿形层序遍历 11](#_Toc142841229)

[全排列 13](#_Toc142841230)

[螺旋打印矩阵 13](#_Toc142841231)

[翻转链表指定区间 14](#_Toc142841232)

[删除排序链表中的重复元素Ⅰ-保留 14](#_Toc142841233)

[删除排序链表中的重复元素II-不保留 14](#_Toc142841234)

[最长递增子序列 16](#_Toc142841235)

[环形链表II-返回入环节点 16](#_Toc142841236)

[接雨水 17](#_Toc142841237)

[二叉树中的最大路径和 17](#_Toc142841238)

[重排链表 18](#_Toc142841239)

[删除链表的倒数第N个结点 20](#_Toc142841240)

[二叉树的中序遍历 20](#_Toc142841241)

[二叉树的右视图 22](#_Toc142841242)

[用栈实现队列 22](#_Toc142841243)

[用队列实现栈 23](#_Toc142841244)

[最长公共子序列 23](#_Toc142841245)

[最长公共子数组 24](#_Toc142841246)

[编辑距离 24](#_Toc142841247)

[合并区间 25](#_Toc142841248)

[寻找两个正序数组的中位数 25](#_Toc142841249)

[智能指针-sharedptr 26](#_Toc142841250)

[智能指针-uniquePtr 27](#_Toc142841251)

[复原 IP 地址 28](#_Toc142841252)

[下一个排列 29](#_Toc142841253)

[排序链表-归并 30](#_Toc142841254)

[排序链表-快排 31](#_Toc142841255)

[爬楼梯-O(1) 32](#_Toc142841256)

[最小花费爬楼梯 32](#_Toc142841257)

[x 的平方根 32](#_Toc142841258)

[链表两数相加 33](#_Toc142841259)

[括号生成-n对括号的所有有效组合 33](#_Toc142841260)

[滑动窗口最大值 34](#_Toc142841261)

[比较版本号 35](#_Toc142841262)

[子集问题 35](#_Toc142841263)

[从前序与中序遍历序列构造二叉树 36](#_Toc142841264)

[最长有效括号长度 36](#_Toc142841265)

[零钱兑换-完全背包 37](#_Toc142841266)

[0-1背包模板 37](#_Toc142841267)

[字符串相乘 38](#_Toc142841268)

[最小栈 38](#_Toc142841269)

[剑指offer 40](#_Toc142841270)

[03数组中的重复数组 40](#_Toc142841271)

[04 二维数组中的查找 40](#_Toc142841272)

[05 替换空格 41](#_Toc142841273)

[06 从尾到头打印链表 41](#_Toc142841274)

[07 前序中序重建二叉树 42](#_Toc142841275)

[09用两个栈实现队列 42](#_Toc142841276)

[10- I斐波那契数列 43](#_Toc142841277)

[10- II青蛙跳台阶问题 43](#_Toc142841278)

[11旋转数组的最小数字 43](#_Toc142841279)

[12矩阵中的路径 44](#_Toc142841280)

[58 左旋字符串 44](#_Toc142841281)

[随想录 45](#_Toc142841282)

[Others 45](#_Toc142841283)

[求一个数组的所有连续子数组(索引连续) 45](#_Toc142841284)

# TopCode

## 无重复字符的最长子串

给定一个字符串 s ，找出其中不含有重复字符的 最长子串的长度。

思路：使用哈希表加滑动窗口，检测到重复元素则删除start元素直到找不到该元素。

int lengthOfLongestSubstring(string s) {

    // 哈希 + 滑动窗口

    unordered\_set<char> uset;

    int start = 0;

    int result = 0;

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

        while(uset.find(s[end]) != uset.end()){

            uset.erase(s[start]);

            start++;

        }

        result = max(result, end - start + 1);

        uset.insert(s[end]);

    }

    return result;

}

## 反转链表

给你单链表的头节点 head ，请你反转链表，并返回反转后的链表。

思路：pre, cur双指针迭代到最后

ListNode\* reverseList(ListNode\* head) {

    ListNode\* pre = nullptr;

    ListNode\* cur = head;

    while(cur){

        ListNode\* tmp = cur->next;

        cur->next = pre;

        pre = cur;

        cur = tmp;

    }

    return pre;

}

## LRU 缓存

设计并实现一个满足LRU(最近最少使用)缓存约束的数据结构。

class LRUCache {

public:

    LRUCache(int capacity) : \_capacity(capacity){}

    int get(int key) {

        auto it = umap.find(key);

        if(it == umap.end()) return -1;

        int result = it->second->second;

        put(key, result);

        return result;

    }

    void put(int key, int value) {

        auto it = umap.find(key);

        if(it != umap.end()){

            chache.erase(it->second);

            umap.erase(key);

        }

        chache.push\_front(pair<int, int>(key, value));

        umap.emplace(key, chache.begin());

        if(chache.size() > \_capacity) {

            umap.erase(chache.back().first);

            chache.pop\_back();

        }

    }

private:

    // list front is fresh, back is old

    int \_capacity;

    list<pair<int, int>> chache;

    unordered\_map<int, list<pair<int, int>>::iterator> umap;

};

## 数组中的第k大的元素

思路：构建大顶堆，依次弹出堆顶的k-1个元素后取堆顶元素

使用快排，返回中间节点的编号后选择性递归

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

    //维护一个大顶堆,然后弹出堆顶k-1个元素

    priority\_queue<int, vector<int>, less<int>> pque;

    for(int i : nums) {

        pque.push(i);

    }

    for(int i = 0; i < k - 1; i++) pque.pop();

    return pque.top();

}

int partition(vector<int>& nums, int l, int r)

{

    int key = nums[l];

    int index\_mid = l;

    for(int j = l + 1; j <= r; j++){

        if(nums[j] > key) swap(nums[++index\_mid], nums[j]);

    }

    nums[l] = nums[index\_mid];

    nums[index\_mid] = key;

    return index\_mid;

}

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

    int l = 0, r = nums.size() - 1;

    while (true)

    {

        int idx = partition(nums, l, r);

        if (idx == k - 1)

            return nums[idx];

        else if (idx < k - 1)

            l = idx + 1;

        else

            r = idx - 1;

    }

}

## K个一组翻转链表

给你链表的头节点 head ，每 k 个节点一组进行翻转，请你返回修改后的链表。

思路：递归每次翻转k个节点，把head指向下一次递归后的结果

ListNode\* reverseKGroup(ListNode\* head, int k) {

    // 递归 + 两两交换

    ListNode\* tail = head;

    for(int i = 0; i < k; i++){

        if(tail == nullptr) return head;

        tail = tail->next;

    }

    ListNode\* pre = nullptr;

    ListNode\* cur = head;

    while(cur != tail){

        ListNode\* tmp = cur->next;

        cur->next = pre;

        pre = cur;

        cur = tmp;

    }

    head->next = reverseKGroup(tail, k);

    return pre;

}

## 三数之和

思路：先排序后遍历单个数字+首尾双指针，去重在指针移动等于的时候去重。

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

    //三数之和，数组可能有重复元素

    vector<vector<int>> result;

    sort(nums.begin(), nums.end());

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

        if(nums[a] > 0) break;

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

        int left = a + 1;

        int right = nums.size() - 1;

        while(left < right){

            if(nums[a] + nums[left] + nums[right] == 0){

                result.push\_back({nums[a], nums[left], nums[right]});

                while(left < right && nums[left] == nums[left+1]) left++;

                while(left < right && nums[right] == nums[right-1]) right--;

                left++;

                right--;

            }

            else if(nums[a] + nums[left] + nums[right] < 0) left++;

            else right--;

        }

    }

    return result;

}

## 最大子数组和

找出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。

思路：dp[i]表示子数组以nums[i]结尾的最大和

int maxSubArray(vector<int>& nums) {

    //dp[i]表示以子数组nums[i]结尾的最大和

    vector<int> dp(nums.size(), 0);

    int res = nums[0];

    dp[0] = nums[0];

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

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

        res = max(res, dp[i]);

    }

    return res;

}

## 快速排序

void quickSort(vector<int>& nums, int left, int right){

    if(left >= right) return;

    // int randomINdex = left + rand() % (right - left);

    // swap(nums[left], nums[randomINdex]);

    int key = nums[left];

    int num\_mid = left;

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

        if(nums[i] < key) swap(nums[i], nums[++num\_mid]);

    }

    nums[left] = nums[num\_mid];

    nums[num\_mid] = key;

    quickSort(nums, left, num\_mid - 1);

    quickSort(nums, num\_mid + 1, right);

}

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

    quickSort(nums, 0, nums.size() - 1);

    return nums;

}

## 归并排序

vector<int> tmp;

void mergesort(vector<int>& nums, int start, int end){

    if(start >= end) return;

    int mid = (start + end) / 2;

    mergesort(nums, start, mid);

    mergesort(nums, mid+1, end);

    int i = start, j = mid + 1;

    int cnt = start;

    while(i <= mid && j <= end){

        if(nums[i] <= nums[j]) tmp[cnt++] = nums[i++];

        else tmp[cnt++] = nums[j++];

    }

    while(i <= mid && j > end) tmp[cnt++] = nums[i++];

    while(j <= end && i > mid) tmp[cnt++] = nums[j++];

    for(int i = start; i <=end; i++){

        nums[i] = tmp[i];

    }

    // copy(tmp.begin() + start, tmp.begin() + end +1, nums.begin() + start);

}

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

    tmp = nums;

    mergesort(nums, 0, nums.size()-1);

    return nums;

}

# 堆排序

void maxHeapify(vector<int>& nums, int i, int len){

    int left = 2\*i + 1;

    int right = 2\*i + 2;

    int largeIndex = i;

    if (left <= len && nums[left] > nums[largeIndex]){

        largeIndex = left;

    }

    if (right <= len && nums[right] > nums[largeIndex]){

        largeIndex = right;

    }

    if(largeIndex != i){

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

        maxHeapify(nums, largeIndex, len);

    }

}

void buildMaxHeap(vector<int>& nums, int len){

    for(int i = len/2; i>=0; i--){

        maxHeapify(nums, i, len);

    }

}

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

    int len = nums.size() - 1;

    buildMaxHeap(nums, len);

    for(int i = len; i >=1; i--){

        swap(nums[i], nums[0]); //弹出堆顶元素

        len--;

        maxHeapify(nums, 0, len);//维护大顶堆

    }

    return nums;

}

## 合并两个有序链表

思路：两个链表比较直到有一个为空退出，退出后直接接上。

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

    ListNode\* dummyHead = new ListNode(INT\_MAX);

    ListNode\* cur = dummyHead;

    while(list1 && list2){

        if(list1->val < list2->val) {

            cur->next = list1;

            list1 = list1->next;

        }else{

            cur->next = list2;

            list2 = list2->next;

        }

        cur = cur->next;

    }

    if(list1 == nullptr && list2 != nullptr) cur->next = list2;

    if(list1 != nullptr && list2 == nullptr) cur->next = list1;

    return dummyHead->next;

}

## 合并K个有序链表

思路：使用优先队列把每个链表的非空头节点加入，每次弹出最小的元素，插入该元素的next直到队列为空

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

    auto cmp = [&](const ListNode\* a, const ListNode\* b) {

        return a->val > b->val;

    };

    priority\_queue<ListNode\*, vector<ListNode\*>, decltype(cmp)> pque(cmp);

    for(ListNode\* node : lists) if(node) pque.push(node);

    ListNode\* dummyHead = new ListNode(INT\_MAX, nullptr);

    ListNode\* cur = dummyHead;

    while(!pque.empty()) {

        ListNode\* node = pque.top();

        pque.pop();

        cur->next = node;

        cur = cur->next;

        if(node->next) pque.push(node->next);

    }

    return dummyHead->next;

}

## 两数之和

在该数组中找出和为目标值 target的那两个整数。

思路：一次遍历，用哈希表存储之前的结果

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

    //使用哈希表存储

    unordered\_map<int, int> umap;

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

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

        if(it != umap.end()){

            return {i, it->second};

        }

        umap.emplace(nums[i], i);

    }

    return {};

}

## 二叉树的层序遍历

思路：用队列取记录每一层，同时每一层用一个固定size去for循环

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

    vector<vector<int>> result;

    queue<TreeNode\*> layerq;

    if(root) layerq.push(root);

    while(!layerq.empty()){

        int size = layerq.size();

        vector<int> layer;

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

            TreeNode\* node = layerq.front();

            layerq.pop();

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

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

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

        }

        result.push\_back(layer);

    }

    return result;

}

## 最长回文子串

找到字符串s中最长的回文子串

思路：dp[i][j]表示s[i]-s[j]是不是回文子串，只有当s[i] == s[j]，且字符串长度<=2或者dp[i+1][j-1]，dp[i][j]=1

string longestPalindrome(string s) {

    // dp[i][j]表示s[i]-s[j]是不是回文子串

    vector<vector<int>> dp(s.size(), vector<int>(s.size(), 0));

    string result;

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

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

            if(s[i] == s[j]) {

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

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

            }

            if(dp[i][j] && result.size() < (j-i+1)) {

                result = s.substr(i, j - i + 1);

            }

        }

    }

    return result;

}

## 搜索旋转排序数组

升序数组在k下标出旋转一次后搜索target是否在数组之中。

思路：两个区间一定有序，当确定在一个有序区间时候可以搜索这个区间，反之搜索另一个区间

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

    //两个子区间一定有一个有序，右>左就说明有序

    int left = 0, right = nums.size() - 1;

    while(left <= right){

        int mid = (left + right)/2;

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

        if(nums[mid] <= nums[right]){ //右区间有序

            if(target > nums[mid] && nums[right] >= target) left = mid + 1;

            else right = mid - 1;

        }

        if(nums[mid] >= nums[left]){ //左区间有序

            if(target < nums[mid] && nums[left] <= target) right = mid - 1;

            else left = mid + 1;

        }

    }

    return -1;

}

## 环形链表

给你一个链表的头节点 head ，判断链表中是否有环。

bool hasCycle(ListNode \*head) {

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

    while(fast && fast->next){

        slow = slow->next;

        fast = fast->next->next;

        if(slow == fast) return true;

    }

    return false;

}

## 买卖股票的最佳时机

只能买卖一次

int maxProfit(vector<int>& prices) {

    // dp[i][0]表示当天不持有 dp[i][1]表示当天持有,现金是多少

    vector<vector<int>> dp(prices.size(), vector<int>(2,0));

    dp[0][0] = 0; dp[0][1] = -prices[0];

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

        dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i]);

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

    }

    return dp[prices.size()-1][0];

}

可以多次买卖

int maxProfit(vector<int>& prices) {

    // dp[i][0]表示当天不持有 dp[i][1]表示当天持有,现金是多少

    vector<vector<int>> dp(prices.size(), vector<int>(2,0));

    dp[0][0] = 0; dp[0][1] = -prices[0];

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

        dp[i][0] = max(dp[i-1][0], dp[i-1][1] + prices[i]);

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

    }

    return dp[prices.size()-1][0];

}

可以多次买卖，但是有冷静期

int maxProfit(vector<int>& prices) {

    if (prices.empty()) return 0;

    //dp[i][0]第i天持有,dp[i][1]第i天不持有,且处于冷静期,dp[i][2]第i天不持有,不处于冷静期

    vector<vector<int>> dp(prices.size(), vector<int>(3, 0));

    dp[0][0] = -prices[0]; dp[0][1] = 0; dp[0][2] = 0;

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

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

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

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

}

return max(dp[prices.size() - 1][1], dp[prices.size() - 1][2]);

}

## 合并两个有序数组

思路：双指针用后面开始往nums1插入，当遇到一个数组为空时直接顺序插入另一个数组

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

    int p1 = m - 1, p2 = n - 1;

    int total = m + n - 1;

    while(p1 >= 0 || p2 >= 0) {

        if(p1 < 0) nums1[total--] = nums2[p2--];

        else if(p2 < 0) nums1[total--] = nums1[p1--];

        else if(nums1[p1] < nums2[p2]) nums1[total--] = nums2[p2--];

        else nums1[total--] = nums1[p1--];

    }

}

## 岛屿数量

思路：找到一个i后就开始dfs他的上下左右并把搜到的陆地变为0

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

    if(i<0 || j<0 || i>=grid.size() ||

        j>=grid[0].size() || grid[i][j]=='0') return;

    grid[i][j]='0';

    dfs(grid,i+1,j);

    dfs(grid,i-1,j);

    dfs(grid,i,j+1);

    dfs(grid,i,j-1);

}

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

    int count = 0;

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

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

            if(grid[i][j] == '1'){

                dfs(grid,i,j);

                count++;

            }

        }

    }

    return count;

}

## 二叉树的最近公共祖先

思路：普通二叉树从底往上搜索，遇到空或者目标节点就把这个节点往上返回

二叉搜索树从上往下搜索，找到在区间里面的节点就是答案。

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

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

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

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

    if(left == NULL && right == NULL) return NULL;

    else if(left != NULL && right == NULL) return left;

    else if(left == NULL && right != NULL) return right;

    else return root;

}

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

    if (root->val > p->val && root->val > q->val) {

        return lowestCommonAncestor(root->left, p, q);

    } else if (root->val < p->val && root->val < q->val) {

        return lowestCommonAncestor(root->right, p, q);

    } else return root;

}

## 二叉树的锯齿形层序遍历

思路：跟普通的层序遍历一样，但是单层的记录使用双端队列，奇偶分别往前端后末端插入

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

    //锯齿层序遍历

    vector<vector<int>> result;

    queue<TreeNode\*> layerq;

    bool left2right = true;

    if(root) layerq.push(root);

    while(!layerq.empty()) {

        int size = layerq.size();

        deque<int> layer;

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

            TreeNode\* node = layerq.front();

            layerq.pop();

            if(left2right) layer.push\_back(node->val);

            else layer.push\_front(node->val);

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

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

        }

        left2right = !left2right;

        result.push\_back(vector<int>(layer.begin(), layer.end()));

    }

    return result;

}

## 全排列

思路：全排列不使用start控制，只使用used数组控制

当有重复元素时需要对nums排序后判断若两个相邻元素相等前一个为used为false则跳过

vector<vector<int>> result;

vector<int> path;

void backtracking(vector<int>& nums,  vector<int> used){

    if(path.size() == nums.size()){

        result.push\_back(path);

        return;

    }

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

        if(used[i] == true) continue;

        path.push\_back(nums[i]);

        used[i] = 1;

        backtracking(nums, used);

        used[i] = 0;

        path.pop\_back();

    }

}

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

    vector<int> used(nums.size(),0);

    backtracking(nums, used);

    return result;

}

## 螺旋打印矩阵

思路：设定四个边界值，四个for循环，边界值相交后退出

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

    vector<int> result;

    int m = matrix.size();  if(m == 0) return result;

    int n = matrix[0].size();

    int left = 0, right = n - 1;

    int up = 0, down = m - 1;

    while(true) {

        for(int i = left; i <= right; i++)  result.push\_back(matrix[up][i]);

        if(++up > down) break;

        for(int i = up; i <= down; i++) result.push\_back(matrix[i][right]);

        if(--right < left) break;

        for(int i = right; i >=left; i--) result.push\_back(matrix[down][i]);

        if(--down < up) break;

        for(int i = down; i >= up; i--) result.push\_back(matrix[i][left]);

        if(++left > right) break;

    }

    return result;

}

## 翻转链表指定区间

思路：利用三个指针pre不变，不断把cur->next节点冒泡到前面

class Solution {

public:

ListNode\* reverseBetween(ListNode\* head, int left, int right) {

    ListNode \*dummyHead = new ListNode(INT\_MAX, head);

    ListNode \*pre = dummyHead;

    for(int i = 0; i < left - 1; i++) pre = pre->next;

    ListNode \*cur = pre->next;

    int gap = right - left;

    for(int i = 0; i < gap; i++) {

        ListNode \*tmp = cur->next;

        cur->next = tmp->next;

        tmp->next = pre->next;

        pre->next = tmp;

    }

    return dummyHead->next;

}

## 删除排序链表中的重复元素Ⅰ-保留

ListNode\* deleteDuplicates(ListNode\* head) {

    if(head == nullptr) return head;

    ListNode\* dummyhead = new ListNode(-1, head);

    ListNode\* cur = head;

    while(cur->next){

        if(cur->val == cur->next->val){

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

        }else cur = cur->next;

    }

    return dummyhead->next;

}

## 删除排序链表中的重复元素II-不保留

思路：只使用一个指针，通过while来删除

ListNode\* deleteDuplicates(ListNode\* head) {

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

    ListNode\* dummyHead = new ListNode(INT\_MAX, head);

    ListNode\* cur = dummyHead;

    while(cur && cur->next && cur->next->next){

        if(cur->next->val == cur->next->next->val){

            int x = cur->next->val;

            while(cur->next && cur->next->val == x) cur->next = cur->next->next;

        }else cur = cur->next;

    }

    return dummyHead->next;

}

**字符串相加**

思路：用两个栈来对齐数字的末端，用一个carry来代表进位，最后需要翻转res

末端双指针也可以实现上面的功能

string addStrings(string num1, string num2) {

    //用两个栈来模拟

    stack<int> st1, st2;

    for(char c : num1) st1.push(c - '0');

    for(char c : num2) st2.push(c - '0');

    string res;

    int carry = 0;

    while(!st1.empty() || !st2.empty()) {

        int a = 0, b = 0;

        if(!st1.empty()) {

            a = st1.top();

            st1.pop();

        }

        if(!st2.empty()) {

            b = st2.top();

            st2.pop(); }

        carry += (a + b);

        res.push\_back(carry % 10 + '0');

        carry /= 10;

    }

    if(carry == 1) res.push\_back('1');

    reverse(res.begin(), res.end());

    return res;

}

string addStrings(string num1, string num2) {

    string res;

    int p1 = num1.size() - 1, p2 = num2.size() - 1;

    int carry = 0;

    while(p1 >=0 || p2 >=0){

        int a = p1 >= 0 ? num1[p1] - '0' : 0;

        int b = p2 >= 0 ? num2[p2] - '0' : 0;

        carry += (a + b);

        res.push\_back(carry % 10 + '0');

        carry /= 10;

        p1--;

        p2--;

    }

    if(carry == 1) res.push\_back('1');

    reverse(res.begin(), res.end());

    return res;

}

## 最长递增子序列

思路：dp[i]表示i之前包括i的以nums[i]结尾的最长递增子序列的长度

int lengthOfLIS(vector<int>& nums) {

if (nums.size() <= 1) return nums.size();

    //dp[i] 表示nums[i]结尾的最长递增子序列

    vector<int> dp(nums.size(), 1);

    dp[0] = 1;

    int result;

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

        for(int j = 0; j < i; j++){

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

        }

        result = max(result, dp[i]);

    }

    return result;

}

## 环形链表II-返回入环节点

思路：快慢指针从head出发，相遇时从head和相遇节点同时出发一个节点(每次移动一个单位)，相遇为入环节点

ListNode \*detectCycle(ListNode \*head) {

    ListNode \*slow = head;

    ListNode \*fast = head;

    while(fast != nullptr && fast->next != nullptr) {

        fast = fast->next->next;

        slow = slow->next;

        if(fast == slow) {

            ListNode \*node = head;

            while(node != slow){

                node = node->next;

                slow = slow->next;

            }

            return slow;

        }

    }

    return nullptr;

}

## 接雨水

思路：left[i]表示下标i位置及其左边的最高柱子的高度，right[i]表示下标 i位置及其右边的最高柱子的高度

int trap(vector<int>& height) {

    int n = height.size();

    vector<int> left(n, 0), right(n, 0);

    left[0] = height[0];

    right[n-1] = height[n-1];

    for(int i = 1; i < n; i++) {

        left[i] = max(left[i-1], height[i]);

        right[n - i -1] = max(height[n-i-1], right[n-i]);

    }

    int ans = 0;

    for(int i = 0; i < n; i++) ans += min(left[i], right[i]) - height[i];

    return ans;

}

## 二叉树中的最大路径和

思路：从下向上递归，返回值是当前节点的最大贡献值，递归的途中记录每个节点为根的最大路径和

int res = INT\_MIN;

int traversal(TreeNode\* root) {

    //返回值是当前节点的最大贡献值

    if(root == nullptr) return 0;

    int left = traversal(root->left);

    int right = traversal(root->right);

    left = left > 0 ? left : 0;

    right = right > 0 ? right : 0;

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

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

}

int maxPathSum(TreeNode\* root) {

    traversal(root);

    return res;

}

## 重排链表

思路：找到中间节点后分成两条链表，翻转后面的链表后依次合并得到新的链表

ListNode\* findMidNode(ListNode\* head) {

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

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

        slow = slow->next;

        fast = fast->next->next;

    }

    return slow;

}

ListNode\* reverseList(ListNode\* head) {

    ListNode\* pre = nullptr;

    ListNode\* cur = head;

    while (cur) {

        ListNode\* tmp = cur->next;

        cur->next = pre;

        pre = cur;

        cur = tmp;

    }

    return pre;

}

void mergeList(ListNode\* l1, ListNode\* l2) {

    // l1 >= l2

    ListNode \*tmp1, \*tmp2;

    while (l1 && l2) {

        tmp1 = l1->next;

        tmp2 = l2->next;

        l1->next = l2;

        l1 = tmp1;

        l2->next = l1;

        l2 = tmp2;

    }

}

void reorderList(ListNode\* head) {

    // 查找中间节点

    ListNode\* mid = findMidNode(head);

    // 一分为二

    ListNode\* l1 = head;

    ListNode\* l2 = mid->next;

    mid->next = nullptr;

    // 反转第二条链表,再合并两条链表

    l2 = reverseList(l2);

    mergeList(l1, l2);

}

## 删除链表的倒数第N个结点

思路：fast先走n个节点，然后fast和slow一起出发，知道fast走到最后，使slow指向要删除的前一个。

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

    ListNode \*dummyHead = new ListNode(INT\_MAX, head);

    ListNode \*fast = dummyHead;

    ListNode \*slow = dummyHead;

    for(int i = 0; i < n; i++) fast = fast->next;

    while(fast->next){

        fast = fast->next;

        slow = slow->next;

    }

    ListNode \*tmp = slow->next;

    slow->next = slow->next->next;

    delete tmp;

    return dummyHead->next;

}

## 二叉树的中序遍历

思路: 左中右，递归很好做，使用迭代法则需要使用栈来进行存储

vector<int> result;

void traversal(TreeNode\* root) {

    if(root == nullptr) return;

    traversal(root->left);

    result.push\_back(root->val);

    traversal(root->right);

}

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

    traversal(root);

    return result;

}

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

    vector<int> result;

    stack<TreeNode\*> st;

    while(root != nullptr || !st.empty()) {

        if(root != nullptr) {

            st.push(root);

            root = root->left;

        }else{

            root = st.top();

            st.pop();

            result.push\_back(root->val);

            root = root->right;

        }

    }

return result; }

## 二叉树的右视图

思路：先访问右边的节点，当depth和resul的size第一次相等时加入数据即为当层的最右边的数据

vector<int> result;

void traversal(TreeNode\* root, int depth) {

    if(root == nullptr) return;

    if(depth == result.size()) result.push\_back(root->val);

    traversal(root->right, depth+1);

    traversal(root->left, depth+1);

}

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

    traversal(root, 0);

    return result;

}

## 用栈实现队列

思路：两个栈干就完了，一个用来进一个用来出

class MyQueue {

public:

    MyQueue() = default;

bool empty() { return st1.empty() && st2.empty(); }

    void push(int x) { st1.push(x); }

    int pop() {

        if(st2.empty()){

            while(!st1.empty()){

                int x = st1.top();

                st1.pop();

                st2.push(x);

            }

        }

        int result = st2.top();

        st2.pop();

        return result;

    }

    int peek() {

        int result = this->pop();

        st2.push(result);

        return result;

    }

private:

    stack<int> st1;

    stack<int> st2;

};

## 用队列实现栈

思路：另一个队列只用来复制，弹出的时候复制队列的前面size-1个元素，最后的就是结果

class MyStack {

public:

    MyStack() {}

    void push(int x) {

        que1.push(x);

    }

    int pop() {

        int size = que1.size() - 1;

        while(size--){

            que2.push(que1.front());

            que1.pop();

        }

        int res = que1.front();

        que1.pop();

        que1 = que2;

        while(!que2.empty()) que2.pop();

        return res;

    }

    int top() {

        return que1.back();

    }

    bool empty() {

        return que1.empty();

    }

private:

    queue<int> que1;

    queue<int> que2;

};

## 最长公共子序列

思路：dp[i][j]：长度为[0, i - 1]的字符串text1与长度为[0, j - 1]的字符串text2的最长公共子序列

int longestCommonSubsequence(string text1, string text2) {

    vector<vector<int>> dp (text1.size() + 1, vector<int>(text2.size() + 1, 0));

    for(int i = 1; i <= text1.size(); i++){

        for(int j = 1; j <= text2.size(); j++){

            if(text1[i-1] == text2[j-1]) dp[i][j] = dp[i-1][j-1] + 1;

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

        }

    }

    return dp[text1.size()][text2.size()];

}

## 最长公共子数组

思路：dp[i][j]：以下标i - 1为结尾的A，和以下标j - 1为结尾的B，最长重复子数组长度为dp[i][j]

int findLength(vector<int>& nums1, vector<int>& nums2) {

    int res = -1;

    vector<vector<int>> dp (nums1.size() + 1, vector<int>(nums2.size() + 1, 0));

    for(int i = 1; i <= nums1.size(); i++){

        for(int j = 1; j <= nums2.size(); j++){

            if(nums1[i-1] == nums2[j-1]) dp[i][j] = dp[i-1][j-1] + 1;

            res = max(res, dp[i][j]);

        }

    }

    return res;

}

## 编辑距离

思路：dp[i][j]表示下i-1结尾的word1，j-1为结尾的word2，最近编辑距离。分三种情况，相等， 增删，换

int minDistance(string word1, string word2) {

    vector<vector<int>> dp(word1.size() + 1, vector<int>(word2.size() + 1, 0));

    for (int i = 0; i <= word1.size(); i++) dp[i][0] = i;

for (int j = 0; j <= word2.size(); j++) dp[0][j] = j;

    for (int i = 1; i <= word1.size(); i++){

        for (int j = 1; j <= word2.size(); j++) {

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

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

            }

            else{

                //增/删

                int tmp1 = min(dp[i-1][j] + 1, dp[i][j-1] + 1);

                //换

                int tmp2 = dp[i - 1][j - 1] + 1;

                dp[i][j] = min(tmp1, tmp2);

            }

        }

    }

    return dp[word1.size()][word2.size()];

}

## 合并区间

思路：第一次进去直接更新left和right，后续需要发现[0]>right再擦汗如结果和更新

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

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

    vector<vector<int>> merged;

    int left = INT\_MAX, right = INT\_MIN;

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

        if(right != INT\_MIN && intervals[i][0] > right){

            merged.push\_back({left, right});

            left = intervals[i][0];

            right = intervals[i][1];

        }

        if(intervals[i][0] < left) left = intervals[i][0];

        if(intervals[i][1] > right) right = intervals[i][1];

    }

    merged.push\_back({left, right});

    return merged;

}

## 寻找两个正序数组的中位数

思路：使用双指针，移动(m+n)/2次，但是每次赋值之前需要保留上一次的结果

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

    int m = nums1.size(), n = nums2.size();

    int i = 0, j = 0, l = 0, r = 0;

    for(int x = 0; x <= (m+n)/2; x++) {

        l = r;

        if(i >= m) r = nums2[j++];

        else if(j >= n) r = nums1[i++];

        else{

            if(nums1[i] > nums2[j]) r = nums2[j++];

            else r = nums1[i++];

        }

    }

    return (m + n) & 1 ? r : (l + r) / 2.0;

}

## 智能指针-sharedptr

template<typename T>

class MysharedPtr{

private:

    T\* \_ptr;

    int\* \_count;

public:

    MysharedPtr(T\* ptr = nullptr): \_ptr(ptr) {

        if(\_ptr == nullptr) \_count = new int(0);

        else \_count = new int(1);

    }

    ~MysharedPtr(){

        if(this->\_ptr){

            if(--(\*this->\_count) == 0){

                delete this->\_count;

                delete this->\_ptr;

                this->\_count = nullptr;

                this->\_ptr = nullptr;

            }

        }

    }

    MysharedPtr(const MysharedPtr& shared){

        this->\_ptr = shared.\_ptr;

        this->\_count = shared.\_count;

        (\*this->\_count)++;

    }

    MysharedPtr& operator=(const MysharedPtr& shared){

        if(this == &shared) return \*this;

        if(this->\_ptr){

                if(--(\*this->\_count) == 0){

                delete this->\_count;

                delete this->\_ptr;

                this->\_count = nullptr;

                this->\_ptr = nullptr;

            }

        }

        this->\_ptr = shared.\_ptr;

        this->\_count = shared.\_count;

        (\*this->\_count)++;

        return \*this;

    }

    T& operator\*() { return \*this->\_ptr;}

    T\* operator->() { return this->\_ptr;}

    int getCount() {return \*this->\_count;}

};

## 智能指针-uniquePtr

template<typename T>

class MyweakPtr {

private:

    T \* \_ptr;

    MyweakPtr(const MyweakPtr &p) = delete;

    MyweakPtr& operator=(const MyweakPtr &p) = delete;

public:

    MyweakPtr(T \* ptr = nullptr) : \_ptr(ptr) {}

    ~MyweakPtr(){

        if(\_ptr) delete \_ptr;

    }

    MyweakPtr(MyweakPtr &&p) : \_ptr(p.\_ptr) {

        p.\_ptr = nullptr;

    }

    MyweakPtr& operator=(MyweakPtr &&p) {

        if(\*this == p && p) {

            \_ptr = p.\_ptr;

            p.\_ptr = nullptr;

        }

        return \*this;

    }

    T& oporator\*() const {return \*\_ptr;}

    T\* oporator&() const {return &\_ptr;}

    T\* get() const {

        return \_ptr;

    }

    T\* release() {

        T\* result = \_ptr;

        \_ptr = nullptr;

        return result;

    }

    void reset(T\* p){

        if(p != \_ptr) {

            if(\_ptr) delete \_ptr;

            \_ptr = p;

        }

    }

    void swap(MyweakPtr &p){

        std::swap(p.\_ptr, \_ptr);

    }

};

## 复原 IP 地址

思路：回溯出每一种可能，回溯的参数需要加上控制开始位置和逗号数量

bool isValid(string &s, int start, int end){

    if(start > end) return false;

    if(s[start] == '0' && start != end) return false;

    int net = 0;

    for(int i = start; i <= end; i++) {

        if(s[i] >'9' || s[i] < '0') return false;

        net = net\*10 + (s[i] - '0');

        if(net > 255) return false;

    }

    return true;

}

vector<string> result;

void traversal(string &s, int startIndex, int dotNum) {

    if(dotNum == 3){

        if(isValid(s, startIndex, s.size() - 1)) {

            result.push\_back(s);

        }

        return;

    }

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

        if(isValid(s, startIndex, i)){

            s.insert(s.begin() + i + 1, '.');

        }

        else break;

        traversal(s, i + 2, dotNum + 1);

        s.erase(s.begin() + i + 1);

    }

}

vector<string> restoreIpAddresses(string s) {

    traversal(s, 0, 0);

    return result;

}

## 下一个排列

思路：见代码

void nextPermutation(vector<int>& nums) {

    //数组升序字典序小，降序字典序大

    //找下一个字典序则把最靠右的升序对拆散

    //把该升序对的小值变成右边大于他的最小值，

    //然后把这个值的右边全部变成升序

    int i = nums.size() - 2;

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

    if(i == -1) {

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

    } else {

        int j = nums.size() - 1;

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

        swap(nums[i], nums[j]);

        reverse(nums.begin() + i + 1, nums.end());

    }

    return;

}

## 排序链表-归并

思路：自顶向下归并

ListNode\* mergeList(ListNode\* head1, ListNode\* head2) {

    ListNode \*dummyHead = new ListNode(-1);

    ListNode \*cur = dummyHead;

    while(head1 && head2) {

        if(head1->val < head2->val) {

            cur->next = head1;

            head1 = head1->next;

        } else {

            cur->next = head2;

            head2 = head2->next;}

        cur = cur->next;

    }

    cur->next = head1 ? head1 : head2;

    return dummyHead->next;

}

ListNode\* sortList(ListNode\* head) {    //归并排序

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

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

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

        slow = slow->next;

        fast = fast->next->next;

    }

    fast = slow->next;

    slow->next = nullptr;

return mergeList(sortList(head), sortList(fast));

}

## 排序链表-快排

ListNode\* quickSort(ListNode\* head ,ListNode\* end){

    if(head == end || head->next == end) return head;

    ListNode \*lhead = head, \*utail = head, \*p = head->next;

    while (p != end){

        ListNode \*next = p->next;

        if(p->val < head->val){//头插

            p->next = lhead;

            lhead = p;

        }else { //尾插

            utail->next = p;

            utail = p;

        }

        p = next;

    }

    utail->next = end;

    ListNode\* node = quickSort(lhead, head);

    head->next =  quickSort(head->next, end);

    return node;

}

ListNode\* sortList(ListNode\* head) {

    return quickSort(head, nullptr);

}

## 爬楼梯-O(1)

int climbStairs(int n) {

    if(n <= 2) return n;

    int dp[2];

    dp[0] = 1; dp[1] = 2;

    int sum = 0;

    for(int i = 3; i <= n; i++){

        sum = dp[0] + dp[1];

        dp[0] = dp[1];

        dp[1] = sum;

    }

    return sum;

}

## 最小花费爬楼梯

思路：dp[i]表示爬上第i阶台阶的最小花费

int minCostClimbingStairs(vector<int>& cost) {

    if(cost.size() < 2) return 0;

    //dp[i]表示爬上第i阶台阶的最小花费

    vector<int> dp(cost.size() + 1, 0);

    dp[0] = 0;

    dp[1] = 0;

    for(int i = 2; i <= cost.size(); i++){

        dp[i] = min(dp[i-1] + cost[i-1], dp[i-2] + cost[i-2]);

    }

    return dp[cost.size()];

}

## x 的平方根

int mySqrt(int x) {

    //二分法查找

    if(x <= 1) return x;

    int left = 0, right = x, ans = 0;

    while(left <= right){

        int mid = left + (right - left)/2;

        if(mid == x/mid) return mid;

        if(mid < x/mid) {

            left = mid + 1;

            ans = mid;

        }else right = mid - 1;

    }

    return ans;

}

## 链表两数相加

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

    ListNode\* result = new ListNode(-1);

    ListNode\* Head = result;

    int sum = 0;   //保留

    int carry = 0; //进位

    while(l1 || l2){

        int x1 = l1 == NULL ? 0 : l1->val;

        int x2 = l2 == NULL ? 0 : l2->val;

        sum = x1 + x2 + carry;

        carry = sum / 10;

        sum = sum % 10;

        result->next = new ListNode(sum);

        result = result->next;

        if (l1) l1 = l1->next;

        if (l2) l2 = l2->next;

    }

    if(carry == 1) result->next = new ListNode(1);

    return Head->next;

}

## 括号生成-n对括号的所有有效组合

思路：回溯，先放左括号，在放右括号，左括号的控制参数是n，右括号的控制参数是left

vector<string> result;

string path;

void bactracking(int left, int right, int n){

    if(path.size() == 2\*n){

        result.push\_back(path);

        return;

    }

    if(left < n) {

        path.push\_back('(');

        bactracking(left + 1, right, n);

        path.pop\_back();

    }

    if(right < left ) {

        path.push\_back(')');

        bactracking(left, right + 1, n);

        path.pop\_back();

    }

}

vector<string> generateParenthesis(int n) {

    bactracking(0, 0, n);

    return result;

}

**字符串转换整数 (atoi)**

int strToInt(string str) {

    int index = 0, n = str.size(), sign = 1, res = 0;

    // 处理前置空格

    while (index < n && str[index] == ' ') ++index;

    // 处理符号

    if (index < n && (str[index] == '+' || str[index] == '-')) {

        sign = str[index++] == '+' ? 1 : -1;

    }

    // 处理数字

    while (index < n && isdigit(str[index])) {

        int digit = str[index++] - '0';

        // 判断是否溢出

        if (res > (INT\_MAX - digit) / 10) return sign == 1 ? INT\_MAX : INT\_MIN;

        res = res \* 10 + digit;

    }

    return res \* sign;

}

## 滑动窗口最大值

思路：自定义单调队列，front最大，维护有可能成为最大值的元素即可

Pop：如果窗口移除的元素value等于单调队列的出口元素，那么队列弹出元素，否则不用任何操作

Push：如果push的元素大于入口元素的数值，将队列入口的元素弹出，直到push的元素小于等于队列入口元素。

class myqueue{

public:

void pop(int x){

    if(!que.empty() && x == que.front()) que.pop\_front(); }

void push(int x){

    while(!que.empty() && x > que.back()) que.pop\_back();

    que.push\_back(x);

}

int front(){ return que.front();}

private:

    deque<int> que;

};

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

    myqueue que;

    vector<int> result;

    for(int i = 0; i < k; i++) que.push(nums[i]);

    result.push\_back(que.front());

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

        que.push(nums[i]);

        que.pop(nums[i-k]);

        result.push\_back(que.front());

    }

    return result;

}

## 比较版本号

int compareVersion(string version1, string version2) {

    int i = 0, j = 0;

    while(i < version1.size() || j < version2.size()) {

        int v1 = 0, v2 = 0;

        for(; i < version1.size(); i++){

            if(version1[i] == '.') break;

            v1 = v1\*10 + (version1[i] - '0');

        }

        ++i;

        for(; j < version2.size(); j++){

            if(version2[j] == '.') break;

            v2 = v2\*10 + (version2[j] - '0');

        }

        ++j;

        if(v1 != v2) return v1 > v2 ? 1 : -1;

    }

    return 0;

}

## 子集问题

vector<vector<int>> result;

vector<int> path;

void backtracking(vector<int>& nums, int startIndex) {

    result.push\_back(path);

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

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

        path.push\_back(nums[i]);

        backtracking(nums, i+1);

        path.pop\_back();

    }

}

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

    backtracking(nums, 0);

    return result;

}

## 从前序与中序遍历序列构造二叉树

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

    if(preorder.size() == 0) return nullptr;

    TreeNode\* root = new TreeNode(preorder[0]);

    if(preorder.size() == 1) return root;

    auto it = find(inorder.begin(), inorder.end(), root->val);

    vector<int> inleft(inorder.begin(), it);

    vector<int> inright(it + 1, inorder.end());

    vector<int> perleft(preorder.begin() + 1, preorder.begin() + 1 + inleft.size());

    vector<int> perright(preorder.begin() + 1 + inleft.size(), preorder.end());

    root->left = buildTree(perleft, inleft);

    root->right = buildTree(perright, inright);

    return root;

}

## 最长有效括号长度

int longestValidParentheses(string s) {

    //dp[i]表示以下标i结尾的最长有效括号长度

    vector<int> dp(s.size(), 0);

    int result = 0;

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

        if(s[i] == '(') dp[i] = 0;

        if(s[i] == ')') {

            if(s[i-1] == '(') {

                dp[i] = (i >= 2 ? dp[i - 2] : 0) + 2;

            }else if(i - dp[i - 1] > 0 && s[i - dp[i-1] - 1] == '(') {

                dp[i] = dp[i-1] + ((i - dp[i - 1]) >= 2 ? dp[i - dp[i - 1] - 2] : 0) + 2;

            }

        }

        result = max(result, dp[i]);

    }

    return result;

}

## 零钱兑换-完全背包

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

    //完全背包问题

    //先遍历物品再遍历背包，都是正序遍历

    // dp[j]表示凑满i金额的最少硬币个数

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

    dp[0] = 0;

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

        for(int j = coins[i]; j <= amount; j++) {

            if(dp[j - coins[i]] == INT\_MAX) continue;

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

        }

    }

    if (dp[amount] == INT\_MAX) return -1;

    return dp[amount];

}

## 0-1背包模板

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

for(int i = 0; i < weight.size(); i++) { // 遍历物品

    for(int j = bagWeight; j >= weight[i]; j--) { // 倒叙遍历背包容量

        dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);

    }

}

## 字符串相乘

string multiply(string num1, string num2) {

    //乘积最长为m+n

    //nums1[i] \*nums2[j]的位置存储在res[i+j+1]

    if (num1 == "0" || num2 == "0") return "0";

    int m = num1.size(), n = num2.size();

    vector<int> res(m + n, 0);

    for(int i = m-1; i >= 0; i--){

        int x = num1[i] - '0';

        for(int j = n-1; j >= 0; j--){

            int y = num2[j] - '0';

            res[i+j+1] += x\*y;

        }

    }

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

        res[i-1] += res[i] / 10;

        res[i] %= 10;

    }

    string result;

    for(int i = 0; i < m+n; i++){

        if(i == 0 && res[i] == 0) continue;

        result.push\_back(res[i] + '0');

    }

    return result;

}

## 最小栈

class MinStack {

private:

    stack<int> st;

    stack<int> min\_st;

public:

    MinStack() {

        min\_st.push(INT\_MAX);

    }

    void push(int val) {

        st.push(val);

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

    }

    void pop() {

        st.pop();

        min\_st.pop();

    }

    int top() { return st.top();}

    int getMin() { return min\_st.top();}

};

# 剑指offer

## 03数组中的重复数组

int findRepeatNumber(vector<int>& nums) {

    // 哈希表

    // unordered\_set<int> uset;

    // for(int i : nums){

    //     if(uset.find(i) != uset.end()) return i;

    //     else uset.insert(i);

    // }

    // return -1;

    //元素都在(0,n-1)范围内，原地交换

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

        //确保该位置被换成i，若元素i不存在则会则会映射其他元素

        while(nums[i] != i){

            if(nums[nums[i]] == nums[i]) return nums[i];

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

        }

    }

    return -1;

}

## 04 二维数组中的查找

bool findNumberIn2DArray(vector<vector<int>>& matrix, int target) {

    //二叉搜索树的查找，左下角为root

    int m = matrix.size(); if(m == 0) return false;

    int n = matrix[0].size();

    int i = 0, j = n-1;

    while(i < m && j >=0){

        if(matrix[i][j] == target) return true;

        else if(matrix[i][j] > target) j--;

        else i++;

    }

    return false;

}

// 二分查找

for(vector<int> row : matrix){

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

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

}

return false;

## 05 替换空格

思路：倒序双指针

string replaceSpace(string s) {

    int count = 0;

    for(char c : s) if(c == ' ') count++;

    int oldSize = s.size();

    int newSize = s.size() + 2 \* count;

    s.resize(newSize);

    while(oldSize >= 0){

        if(s[oldSize] != ' ') s[newSize--] = s[oldSize--];

        else{

            s[newSize--] = '0';

            s[newSize--] = '2';

            s[newSize--] = '%';

            oldSize--;

        }

    }

    return s;

}

## 06 从尾到头打印链表

思路：递归打印

vector<int> res;

void traversal(ListNode\* head){

    if(head == NULL) return;

    traversal(head->next);

    res.push\_back(head->val);

}

vector<int> reversePrint(ListNode\* head) {

    traversal(head);

    return res;

}

## 07 前序中序重建二叉树

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

    //前序 中左右, 中序 左中右

    if(preorder.size() == 0) return nullptr;

    int key = preorder[0];

    TreeNode \* root = new TreeNode(key);

    auto it = find(inorder.begin(), inorder.end(), key);

    vector<int> inleft(inorder.begin(), it);

    vector<int> inright(it + 1, inorder.end());

    vector<int> preleft(preorder.begin() + 1, preorder.begin() + 1 + inleft.size());

    vector<int> preright(preorder.begin() + 1 + inleft.size(), preorder.end());

    root->left = buildTree(preleft, inleft);

    root->right = buildTree(preright, inright);

    return root;

}

## 09用两个栈实现队列

class CQueue {

public:

CQueue() {}

void appendTail(int value) {stkin.push(value);}

int deleteHead() {

    if(stkout.empty()){

        while(!stkin.empty()){

            int x = stkin.top(); stkin.pop();

            stkout.push(x);

        }

    }

    if(stkout.empty()) return -1;

    int res = stkout.top();

    stkout.pop();

    return res;

}

private:

stack<int> stkin;

stack<int> stkout;

};

## 10- I斐波那契数列

int fib(int n) {

    //O(1)动态规划

    if(n <= 1) return n;

    int dp[2] = {0};

    dp[0] = 0;

    dp[1] = 1;

    for(int i = 2; i <= n; i++){

        int tmp = dp[0] + dp[1];

        dp[0] = dp[1];

        dp[1] = tmp % (1000000007);

    }

    return dp[1];

}

## 10- II青蛙跳台阶问题

int numWays(int n) {

    //O(1)动态规划

    if(n <= 1) return 1;

    int pre = 1, cur = 1;

    for(int i = 2; i<=n; i++) {

        int tmp = pre + cur;

        pre = cur;

        cur = tmp % (1000000007);

    }

    return cur;

}

## 11旋转数组的最小数字

int minArray(vector<int>& numbers) {

    //二分法寻找最小值, 二分一定会有一个有序区间

    int left = 0;

    int right = numbers.size() - 1;

    while(left < right){

        int mid = (left + right) / 2;

        //只比较mid和right

        if(numbers[mid] < numbers[right]) right = mid;

        else if(numbers[mid] > numbers[right]) left = mid+1;

        else right--;

    }

    return numbers[left];

}

## 12矩阵中的路径

bool backtracking(vector<vector<char>>& board, string word, int index, int i , int j){

    if(board[i][j] != word[index]) return false;

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

    char ori = board[i][j];

    board[i][j] = '\*';

    if(i+1 < board.size() && backtracking(board, word, index + 1, i + 1, j)||

        j+1 < board[0].size() && backtracking(board, word, index + 1, i, j + 1)||

        j-1 >= 0 && backtracking(board, word, index + 1, i, j - 1)||

        i-1 >= 0 && backtracking(board, word, index + 1, i - 1, j )){

            return true;

        }

    board[i][j] = ori;

    return false;

}

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

    //回溯

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

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

            if(backtracking(board, word, 0, i, j)){

                return true;

            }

        }

    }

    return false;

}

## 58 左旋字符串

string reverseLeftWords(string s, int n) {

    reverse(s.begin(), s.end());

    reverse(s.begin(), s.begin() + (s.size() - n));

    reverse(s.end() - n, s.end());

return s;

// int len = s.size();

// s += s;

// return s.substr(n, len);

}

# 随想录

# Others

## 求一个数组的所有连续子数组(索引连续)

如[1,2,3] 生成[1] [2] [3] [1,2] [2,3] [1,2,3]

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

    vector<vector<int>> result;

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

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

            vector<int> subset;

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

                subset.push\_back(nums[k]);

            }

            result.push\_back(subset);

        }

    }

    return result;

}