C++

两数之和

```cpp
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        vector<int> result;
        unordered_map<int, int> umap;
        for (int i = 0; i < nums.size(); i++) {
            auto it = umap.find(target - nums[i]);
            if (it == umap.end()) {
                umap.insert({nums[i], i});
            } else {
                return {i, it->second};
            }
        }

        return result;
    }
};
```

字母异位词

```cpp
class Solution {
public:
    vector<vector<string>> groupAnagrams(vector<string>& strs) {
        vector<vector<string>> result;
        unordered_map<string, vector<string>> umap;
        //将每个单词排序后插入对应的哈希表的键值对中
        for (int i = 0; i < strs.size(); i++) {
            string s = strs[i];
            sort(s.begin(), s.end());
            umap[s].push_back(strs[i]);
        }

        for (auto i = umap.begin(); i != umap.end(); i++) {
            result.push_back(i->second);
        }

        return result;
    }
};
```

最长连续序列

```cpp
class Solution {
public:
    int longestConsecutive(vector<int>& nums) {
        unordered_set<int> set(nums.begin(), nums.end());   //使用集合去重
        int maxLength = 0;
        for (int num : set) {   //查找最长连续序列
            if (set.find(num - 1) == set.end()) {
                int cur = num;
                int curLength = 1;
                while (set.find(num + 1) != set.end()) {
                    curLength++;
                    num++;
                    cur = num;
                }
```

```
                maxLength = max(maxLength, curLength);
            }
        }

        return maxLength;
    }
};
```

移动零

```
class Solution {
public:
    void moveZeroes(vector<int>& nums) {
        int left = 0;
        for (int i = 0; i < nums.size(); i++) {
            if (nums[i]) {  //右指针不为0则与左指针交换
                swap(nums[i], nums[left]);
                left++;     //左指针左侧的数都为0
            }
        }
    }
};
```

盛最多水的容器

```
class Solution {
public:
    int maxArea(vector<int>& height) {
        int left = 0, right = height.size() - 1;
        int res = 0;
        while (left < right) {
            int cur = min(height[left], height[right]) * (right - left);
            res = max(cur, res);
            //容器容量与最短的边有关，往中间靠拢寻找更长的边
            if (height[left] < height[right]) {
                left++;
            } else {
                right--;
            }
        }

        return res;
    }
};
```

三数之和

```
class Solution {    //代码随想录
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        vector<vector<int>> result;
        sort(nums.begin(), nums.end());

        for (int i = 0; i < nums.size() - 2; i++) {
            if (nums[i] > 0) return result;
            if (i > 0 && nums[i] == nums[i - 1]) continue;  //对左指针去重
            int j = i + 1;
            int k = nums.size() - 1;
            while (j < k) {
                vector<int> path;
                if (nums[i] + nums[j] + nums[k] < 0) j++;
```

```
                else if(nums[i] + nums[j] + nums[k] > 0) k--;
                else {
                    path.push_back(nums[i]);
                    path.push_back(nums[j]);
                    path.push_back(nums[k]);
                    result.push_back(path);
                    while (k > j && nums[k] == nums[k - 1]) k--; //对中间去重
                    while (k > j && nums[j] == nums[j + 1]) j++;
                    k--;    //更新值
                    j++;
                    continue;
                }
            }
        }

        return result;
    }
};

class Solution {    //虎哥
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        vector<vector<int>> result;
        sort(nums.begin(), nums.end());
        for (int i = 0; i < nums.size() - 2; i++) {
            if (nums[i] > 0) continue;
            if (i > 0 && nums[i] == nums[i - 1]) continue;
            int left = i + 1;
            int right = nums.size() - 1;
            while (left < right) {
                if (nums[i] + nums[left] + nums[right] < 0) {
                    left++;
                } else if (nums[i] + nums[left] + nums[right] > 0) {
                    right--;
                } else {
                    vector<int> path;
                    path.push_back(nums[i]);
                    path.push_back(nums[left]);
                    path.push_back(nums[right]);
                    result.push_back(path);
                    left++;
                    while (left < right && nums[left] == nums[left - 1]) {
                        left++;
                    }
                }
            }
        }

        return result;
    }
};
```

接雨水

```
class Solution {
public:
    int trap(vector<int>& height) {
        stack<int> st;  //单调栈
        st.push(0);
        int ans = 0;
        for (int i = 1; i < height.size(); i++) {
```

```
            if (height[i] <= height[st.top()]) {
                st.push(i);
            } else {
                while (!st.empty() && height[i] >= height[st.top()]) {
                    int mid = st.top();     //雨水是有底的
                    st.pop();
                    if (!st.empty()) {
                        int left = st.top();
                        int right = i;
                        int h = min(height[right], height[left]) - height[mid];
                        int w = right - left - 1;
                        ans += w * h;
                    }
                }
                st.push(i);
            }
        }

        return ans;
    }
};
```

无重复的最长子串

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        unordered_set<char> set;
        int ans = 0;
        int left = 0;
        for (int right = 0; right < s.size(); right++) {
            while (set.find(s[right]) != set.end()) {
                set.erase(s[left]);
                left++;
            }

            set.insert(s[right]);
            ans = max(ans, right - left + 1);   //根据左右指针位置计算子串长度
        }

        return ans;
    }
};
```

找到字符串中所有字母异位词

```
class Solution {
public:
    vector<int> findAnagrams(string s, string p) {
        vector<int> result;

        if (s.size() < p.size()) {
            return result;
        }
        vector<int> pCout(26, 0);
        vector<int> sCout(26, 0);
        for (int i = 0; i < p.size(); i++) {
            pCout[p[i] - 'a']++;
            sCout[s[i] - 'a']++;
        }
        //比较两个vector是否相等
```

```
        if (pCout == sCout) result.push_back(0);

        for (int i = p.size(); i < s.size(); i++) {
            sCout[s[i] - 'a']++;
            sCout[s[i - p.size()] - 'a']--;
            if (sCout == pCout) result.push_back(i - p.size() + 1);
        }

        return result;
    }
};
```

和为K的子数组

```
class Solution {
public:
    int subarraySum(vector<int>& nums, int k) {
        int ans = 0;
        unordered_map<int, int> umap;   //哈希
        umap[0] = 1;
        int count = 0;
        for (int i = 0; i < nums.size(); i++) {
            count += nums[i];   //前缀和
            if (umap[count - k]) {
                ans += umap[count - k];
            }
            umap[count]++;
        }

        return ans;
    }
};
```

滑动窗口最大值

```
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        vector<int> result;
        deque<int> que; //单调队列
        for (int i = 0; i < k; i++) {
            while (!que.empty() && nums[i] > nums[que.back()]) {    //维护队尾是递减的
                que.pop_back();
            }

            que.push_back(i);
        }
        result.push_back(nums[que.front()]);            // 队首始终是当前窗口的最大值的
下标
        for (int i = k; i < nums.size(); i++) {
            while (!que.empty() && i - que.front() + 1 > k) {       //超出队列长度弹出
                que.pop_front();
            }

            while (!que.empty() && nums[i] > nums[que.back()]) {    //维护队尾是递减的
                que.pop_back();
            }

            que.push_back(i);
            result.push_back(nums[que.front()]);
```

```
        }

        return result;
    }
};
```

最小覆盖子串

```
class Solution {
public:
    string minWindow(string s, string t) {
        if (s.size() == 0 || s.size() < t.size()) {
            return "";
        }
        int match = 0;
        unordered_map<char, int> tmap;
        unordered_map<char, int> smap;
        for (int i = 0; i < t.size(); i++) {    //记录字符出现的频率
            tmap[t[i]]++;
        }

        int tCount = tmap.size();       //注意字符重复的情况要用哈希表去重 "aa"
        int left = 0;
        int right = 0;
        int startIndex = 0, strLen = INT_MAX;
        while (right < s.size()) {
            if (tmap[s[right]]) {
                smap[s[right]]++;
                if (smap[s[right]] == tmap[s[right]]) { //记录出现频率
                    match++;
                }
            }

            while (match == tCount) {
                if (right - left + 1 < strLen) {    //更新结果
                    strLen = right - left + 1;
                    startIndex = left;
                }

                if (tmap[s[left]]) {
                    smap[s[left]]--;        //满足条件时移动左指针
                    if (tmap[s[left]] > smap[s[left]]) {
                        match--;
                    }
                }
                left++;
            }
            right++;
        }

        if (strLen == INT_MAX) {
            return "";
        }
        return s.substr(startIndex, strLen);
    }
};
```

最大子数组和

```
class Solution {
public:
```

```
    int maxSubArray(vector<int>& nums) {
        vector<int> dp(nums.size(), 0);     //dp数组
        dp[0] = nums[0];
        int result = nums[0];
        for (int i = 1; i < nums.size(); i++) {
            dp[i] = max(nums[i], nums[i] + dp[i - 1]);
            result = max(dp[i], result);
        }

        return result;
    }
};
```

合并区间

```
class Solution {
public:
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        vector<vector<int>> result;
        sort(intervals.begin(), intervals.end());
        for (int i = 1; i < intervals.size(); i++){
            if (intervals[i][0] <= intervals[i - 1][1]) {   //覆盖则合并区间
                intervals[i][0] = intervals[i - 1][0];
                intervals[i][1] = max(intervals[i][1], intervals[i - 1][1]);
            } else {
                result.push_back(intervals[i - 1]);         //不覆盖添加到结果中
            }
        }

        result.push_back(intervals[intervals.size() - 1]);

        return result;
    }
};
```

轮转数组

```
class Solution {
public:
    void rotate(vector<int>& nums, int k) {
        k = k % nums.size();        //防止反转时候下标溢出
        if (nums.size() <= 1) {
            return;
        }
        reverse(nums.begin(), nums.end());      //三次反转实现轮转
        reverse(nums.begin(), nums.begin() + k);
        reverse(nums.begin() + k, nums.end());
    }
};
```

除自身以外数组的乘积

```
class Solution {
public:
    vector<int> productExceptSelf(vector<int>& nums) {
        vector<int> pre(nums.size(), 1);
        vector<int> suf(nums.size(), 1);
        vector<int> result(nums.size(), 1);
        int count = 1;
        for (int i = 1; i < nums.size(); i++) { //前缀积
            count = count * nums[i - 1];
```

```
            pre[i] = count;
        }
        count = 1;
        for (int i = nums.size() - 2; i >= 0; i--) {    //后缀积
            count = count * nums[i + 1];
            suf[i] = count;
        }

        for (int i = 0; i < nums.size(); i++) { //前缀积*后缀积
            result[i] = pre[i] * suf[i];
        }

        return result;
    }
};
```

缺失的第一个正数

```
class Solution {
public:
    int firstMissingPositive(vector<int>& nums) {
        for (int i = 0; i < nums.size(); i++) {
            while (nums[i] > 0 && nums[i] < nums.size() &&
            nums[i] != nums[nums[i] - 1]) {          //防止重复元素导致死循环
                swap(nums[i], nums[nums[i] - 1]);   //原地交换数组
            }
        }

        for (int i = 0; i < nums.size(); i++) {
            if (nums[i] != i + 1) {
                return i + 1;            //查找数组缺失的位置
            }
        }

        return nums.size() + 1;          //数组被填满了返回数组数量+1
    }
};
```

矩阵置零

```
class Solution {
public:
    void setZeroes(vector<vector<int>>& matrix) {
        int rowZero = 1;    //0,0位置的行列均要单独判断是否置0
        int colZero = 1;
        int row = matrix.size();
        int col = matrix[0].size();
        if (matrix[0][0] == 0) {
            rowZero = 0;
            colZero = 0;
        }
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                if (i == 0 && j == 0) {
                    continue;
                } else {
                    if (matrix[i][j] == 0) {
                        if (i == 0) {
                            rowZero = 0;
                            matrix[0][j] = 0;
                        } else if (j == 0) {
```

```
                        matrix[i][0] = 0;
                        colZero = 0;
                    } else {
                        matrix[0][j] = 0;
                        matrix[i][0] = 0;
                    }
                }
            }
        }

        for (int i = col - 1; i > 0; i--) {
            if (matrix[0][i] == 0) {
                for (int j = 0; j < row; j++) {
                    matrix[j][i] = 0;
                }
            }
        }

        for (int i = row - 1; i > 0; i--) {
            if (matrix[i][0] == 0) {
                for (int j = 0; j < col; j++) {
                    matrix[i][j] = 0;
                }
            }
        }

        if (rowZero == 0) {
            for (int i = 0; i < col; i++) {
                matrix[0][i] = 0;
            }
        }

        if (colZero == 0) {
            for (int i = 0; i < row; i++) {
                matrix[i][0] = 0;
            }
        }
    }
};
```

螺旋矩阵

```
class Solution {
public:
    vector<int> spiralOrder(vector<vector<int>>& matrix) {
        vector<int> result;
        int rows = matrix.size();
        int cols = matrix[0].size();
        int top = 0;
        int bottom = rows - 1;
        int left = 0;
        int right = cols - 1;
        while (top <= bottom && left <= right) {
            //打印上边
            for (int i = left; i <= right; i++) {
                result.push_back(matrix[top][i]);
            }
            top++;
            if (top > bottom) { //判断是否终止打印
                break;
```

```
            }
            //打印右边
            for (int i = top; i <= bottom; i++) {
                result.push_back(matrix[i][right]);
            }
            right--;
            if (left > right) { //判断是否终止打印
                break;
            }
            //打印下边
            for (int i = right; i >= left; i--) {
                result.push_back(matrix[bottom][i]);
            }
            bottom--;
            //打印左边
            for (int i = bottom; i >= top; i--) {
                result.push_back(matrix[i][left]);
            }
            left++;
        }

        return result;
    }
};
```

旋转图像

```
class Solution {
public:
    void rotate(vector<vector<int>>& matrix) {
        int rows = matrix.size();
        int cols = matrix[0].size();
        int top = 0;
        int bottom = rows - 1;
        int left = 0;
        int right = cols - 1;
        while (left <= right && top <= bottom) {
            //开始旋转
            for (int j = left; j < right; j++) {
                int i = j - left;
                int temp = matrix[top][left + i];       //保存左上位置
                matrix[top][left + i] = matrix[bottom - i][left];
                matrix[bottom - i][left] = matrix[bottom][right - i];
                matrix[bottom][right - i] = matrix[top + i][right];
                matrix[top + i][right] = temp;
            }

            top++;
            bottom--;
            left++;
            right--;
        }
    }
};
```

搜索二维矩阵 II

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int rows = matrix.size();
```

```
        int cols = matrix[0].size();
        int num = matrix[0][cols - 1];
        int top = 0;
        int right = cols - 1;
        while (right >= 0 && top <= rows - 1) { //二分查找
            if (matrix[top][right] > target) {
                right--;
            } else if (matrix[top][right] < target) {
                top++;
            } else {
                return true;
            }
        }

        return false;
    }
};
```

相交链表

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        ListNode* h1 = headA;
        ListNode* h2 = headB;
        while (h1 != h2) {  //若链表不相交，h1=h2=null
            if (!h1) {
                h1 = headB;
            } else {
                h1 = h1->next;
            }

            if (!h2) {
                h2 = headA;
            } else {
                h2 = h2->next;
            }
        }

        return h1;
    }
};
```

反转链表

```
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* node = head;
        ListNode* last = nullptr;
        while (node) {
            ListNode* tmp = node->next;
            node->next = last;
            last = node;
            node = tmp;
        }

        return last;
    }
};
```

回文链表

```
class Solution {
public:
    bool isPalindrome(ListNode* head) {
        ListNode* dummy = new ListNode(0, head);
        ListNode* left = dummy;
        ListNode* right = head;
        while (right && right->next) {
            left = left->next;
            right = right->next->next;
        }

        ListNode* second = left->next;  //找到后半段的起始节点
        second = reverse(second);       //翻转链表

        ListNode* first = head;
        while (first && second) {
            if (first->val != second->val) {
                return false;
            }

            first = first->next;
            second = second->next;
        }

        return true;
    }

    ListNode* reverse(ListNode* head) {
        ListNode* last = nullptr;
        ListNode* node = head;
        while (node) {
            ListNode* tmp = node->next;
            node->next = last;
            last = node;
            node = tmp;
        }

        return last;
    }
};
```

环形链表

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode* fast = head;
        ListNode* slow = head;
        while (fast && fast->next) {    //通过快慢指针判断是否存在环
            fast = fast->next->next;
            slow = slow->next;
            if (fast == slow) {
                return true;
            }
        }

        return false;
    }
};
```

环形链表 II

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode* fast = head;
        ListNode* slow = head;
        if (!head || !head->next) {      //判断头节点
            return NULL;
        }
        while (fast && fast->next){
            fast = fast->next->next;
            slow = slow->next;

            if (fast == slow) {
                break;
            }
        }

        if (fast != slow) {      //快慢指针没有相遇 说明无环
            return NULL;
        }

        fast = head;             //寻找环入口

        while (fast != slow) {
            fast = fast->next;
            slow = slow->next;
        }

        return fast;
    }
};
```

合并两个有序链表

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        ListNode* dummy = new ListNode();
        ListNode* node = dummy;
        ListNode* l1 = list1;
        ListNode* l2 = list2;
        while (l1 && l2) {  //合并链表
            if (l1->val < l2->val) {
                node->next = l1;
                l1 = l1->next;
            } else {
                node->next = l2;
                l2 = l2->next;
            }

            node = node->next;
        }
        //将剩下的链表节点合并
        if (l1) {
            node->next = l1;
        }

        if (l2) {
            node->next = l2;
```

```
        }

        return dummy->next;
    }
};
```

两数相加

```
class Solution {
public:
    ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
        ListNode* dummy = new ListNode();
        ListNode* node = dummy;
        int carry = 0;
        while (l1 || l2 || carry) { //存在进位或l1、l2仍不为空
            int sum1 = 0, sum2 = 0;
            int sum = 0;
            if (l1) {
                sum1 = l1->val;
                l1 = l1->next;
            }

            if (l2) {
                sum2 = l2->val;
                l2 = l2->next;
            }

            if (carry) {        //存在进位
                sum = sum1 + sum2 + carry;
            } else {
                sum = sum1 + sum2;
            }

            node->next = new ListNode(sum % 10);    //新建节点
            node = node->next;
            carry = sum / 10;
        }

        return dummy->next;
    }
};
```

删除链表的倒数第N个结点

```
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode* dummy = new ListNode(0, head);
        ListNode* left = dummy;
        ListNode* right = left;
        int count = n + 1;
        while (count) {     //right比left向后移n+1位
            right = right->next;
            count--;
        }

        while (right) {     //删除结点
            left = left->next;
            right = right->next;
        }
```

```
        left->next = left->next->next;

        return dummy->next;

    }
};
```

两两交换链表中的节点

```
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        ListNode* cur = head;
        ListNode* dummy = new ListNode(0, head);
        ListNode* pre = dummy;
        while (cur && cur->next) {  //pre cur nxt
            ListNode* nxt = cur->next;
            ListNode* tmp = nxt->next;
            pre->next = nxt;
            nxt->next = cur;
            cur->next = tmp;
            pre = cur;      //更新pre为cur结点
            cur = tmp;      //更新cur为nxt的下一个结点
        }

        return dummy->next;
    }
};
```

K个一组翻转链表

```
class Solution {
public:
    ListNode* reverseKGroup(ListNode* head, int k) {
        ListNode* dummy = new ListNode(0, head);
        ListNode* groupPre = dummy; //每组的上一个位置
        ListNode* cur = groupPre;
        ListNode* tmp = nullptr;
        while (true) {
            int ki = k;
            while (ki && cur) {     //寻找下一组的起点
                ki--;
                cur = cur->next;
            }

            if (!cur) {
                break;
            }

            ListNode* groupNxt = cur->next; //记录下一组起点
            ListNode* pre = groupNxt;       //连接本组第一个结点
            cur = groupPre->next;
            while (cur != groupNxt) {   //翻转本组结点
                tmp = cur->next;
                cur->next = pre;
                pre = cur;
                cur = tmp;
            }

            tmp = groupPre->next;
            groupPre->next = pre;
```

```
            groupPre = tmp;
            cur = groupPre;
        }

        return dummy->next;
    }
};
```

随机链表的复制

```
class Solution {
public:
    Node* copyRandomList(Node* head) {
        unordered_map<Node*, Node*> umap;    //深拷贝n个全新的节点
        Node* cur = head;
        while (cur) {
            Node* newNode = new Node(cur->val);
            umap[cur] = newNode;
            cur = cur->next;
        }

        cur = head;
        while (cur) {
            Node* newNode = umap[cur];
            newNode->next = umap[cur->next];
            newNode->random = umap[cur->random];
            cur = cur->next;
        }

        return umap[head];
    }
};
```

排序链表

```
class Solution {
public:
    ListNode* sortList(ListNode* head) {
        if (!head || !head->next) {
            return head;
        }

        ListNode* dummy = new ListNode(0, head);
        ListNode* first = dummy;
        ListNode* second = head;
        while (second && second->next) {    //将链表分别两部分
            first = first->next;
            second = second->next->next;
        }

        second = first->next;
        first->next = NULL;     //第一段尾结点置为空
        first = head;
        ListNode* firstList = sortList(first);  //两两进行排序
        ListNode* secondList = sortList(second);

        return mergeList(firstList, secondList);
    }

    ListNode* mergeList(ListNode* node1, ListNode* node2){
        ListNode* h1 = node1;
```

```
        ListNode* h2 = node2;
        ListNode* dummy = new ListNode();
        ListNode* node = dummy;
        while (h1 && h2) {
            if (h1->val < h2->val) {
                node->next = new ListNode(h1->val);
                h1 = h1->next;
            } else {
                node->next = new ListNode(h2->val);
                h2 = h2->next;
            }

            node = node->next;
        }

        if (h1) {
            node->next = h1;
        }

        if (h2) {
            node->next = h2;
        }

        return dummy->next;
    }
};
```

合并K个升序链表

```
class Solution {
public:
    ListNode* mergeKLists(vector<ListNode*>& lists) {
        if (lists.size() == 0) {
            return NULL;
        } else if (lists.size() == 1) {
            return lists[0];
        }

        while (lists.size() > 1) {  //直到只剩一个链表说明合并结束
            vector<ListNode*> tempLists;
            for (int i = 0; i < lists.size(); i += 2) {
                ListNode* l1 = lists[i];
                ListNode* l2 = NULL;
                if (i + 1 < lists.size()) {
                    l2 = lists[i + 1];
                }

                tempLists.push_back(mergeList(l1, l2)); //两两合并
            }

            lists = tempLists;
        }

        return lists[0];
    }

    ListNode* mergeList(ListNode* h1, ListNode* h2) {
        ListNode* dummpy = new ListNode();
        ListNode* tail = dummpy;
        while (h1 && h2) {
            if (h1->val < h2->val) {
```

```
                tail->next = new ListNode(h1->val);
                h1 = h1->next;
            } else {
                tail->next = new ListNode(h2->val);
                h2 = h2->next;
            }

            tail = tail->next;
        }

        if (h1) {
            tail->next = h1;
        }

        if (h2) {
            tail->next = h2;
        }

        return dummpy->next;
    }
};
```

LRU缓存

```
class node {    //双向链表节点
public:
    node* next;
    node* pre;
    int key;
    int val;
    node (int k, int v) {
        key = k;
        val = v;
        next = NULL;
        pre = NULL;
    }
};

class LRUCache {    //双向链表+哈希表实现LRU缓存
public:
    LRUCache(int capacity) {
        cap = capacity;
        head = new node(0, 0);
        tail = new node(0, 0);
        head->next = tail;
        tail->pre = head;
    }

    int get(int key) {
        if (umap.count(key)) {
            remove(umap[key]);
            headInsert(umap[key]);
            return umap[key]->val;
        } else {
            return -1;
        }
    }

    void put(int key, int value) {
        if (umap.count(key)) {
            remove(umap[key]);
```

```
            delete(umap[key]);
            umap[key] = NULL;
        }

        node* newnode = new node(key, value);
        headInsert(newnode);
        umap[key] = newnode;

        if (umap.size() > cap) {
            node* last = tail->pre;
            remove(last);
            umap.erase(last->key);
            delete last;
        }
    }
    //删除节点
    void remove(node* cur) {
        node* last = cur->pre;
        node* nxt = cur->next;
        last->next = nxt;
        nxt->pre = last;
    }
    //头部插入一个节点
    void headInsert(node* cur) {
        node* tmp = head->next;
        head->next = cur;
        cur->pre = head;
        cur->next = tmp;
        tmp->pre = cur;
    }

private:
    int cap;
    unordered_map<int, node*> umap;
    node* head;
    node* tail;
};
```

二叉树的中序遍历

```
class Solution {
public:

    void traversal(TreeNode* node, vector<int>& result) {
        if (node == NULL) {
            return;
        }

        traversal(node->left, result);
        result.push_back(node->val);    //中序遍历
        traversal(node->right, result);
    }

    vector<int> inorderTraversal(TreeNode* root) {
        vector<int> result;
        if (!root) return result;
        traversal(root, result);
        return result;
    }
```

```
};
```

二叉树的最大深度

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        queue<TreeNode*> que;       //用队列模拟层序遍历
        int result = 0;
        if (!root) {
            return result;
        }

        que.push(root);
        while (!que.empty()) {
            int size = que.size();
            for (int i = 0;  i < size; i++) {
                TreeNode* node = que.front();
                que.pop();
                if (node->left) que.push(node->left);
                if (node->right) que.push(node->right);
            }

            result++;
        }

        return result;
    }
};
```

翻转二叉树

```
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if (!root) {        //终止条件
            return root;
        }

        swap(root->left, root->right);  //前序遍历
        invertTree(root->left);
        invertTree(root->right);

        return root;
    }
};
```

对称二叉树

```
class Solution {
public:
    bool isSymmetric(TreeNode* root) {
        if (!root) {
            return true;
        }

        return compare(root->left, root->right);
    }
    //判断左右子树是否对称
    bool compare(TreeNode* left, TreeNode* right) {
        //终止条件
```

```
        if (left == NULL && right == NULL) {
            return true;
        } else if (left != NULL && right == NULL) {
            return false;
        } else if (left == NULL && right != NULL) {
            return false;
        } else if (left->val != right->val) {
            return false;
        }
        //本层逻辑  比较外侧和内侧是否对称
        bool outside = compare(left->left, right->right); //外侧
        bool inside = compare(left->right, right->left);  //内测
        bool isSame = outside && inside;    //后序遍历
        return isSame;
    }
};
```

二叉树的直径

```
class Solution {
public:
    int diameterOfBinaryTree(TreeNode* root) {
        int res = 0;
        dfs(root, res);
        return res;
    }

    int dfs(TreeNode* root, int&  res) {
        if (!root) {    //终止条件
            return 0;
        }
        //本层逻辑
        int leftSide = dfs(root->left, res);
        int rightSide = dfs(root->right, res);
        res = max(res, leftSide + rightSide);//最大直径为左节点边数+右节点边数
        return max(leftSide, rightSide) + 1; //返回当前边数给上一节点
    }
};
```

将有序数组转换为二叉搜索树

```
class Solution {
public:
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        if (nums.size() == 0) {
            return NULL;
        }

        return buildTree(nums, 0, nums.size() - 1);
    }
    //输入构造数组的索引
    TreeNode* buildTree(vector<int>& nums, int left, int right) {
        if (left > right) {
            return NULL;
        }
        int mid = (right - left) / 2 + left;
        TreeNode* node = new TreeNode(nums[mid]);
        node->left = buildTree(nums, left, mid - 1);
        node->right = buildTree(nums, mid + 1, right);

        return node;
```

```
    }
};
```

验证二叉搜索树

```
class Solution {
public:
    bool isValidBST(TreeNode* root) {   //中序遍历判断数组是否为递增的
        vector<int> result;
        track(root, result);
        for (int i = 1; i < result.size(); i++) {
            if (result[i] <= result[i - 1]) {
                return false;
            }
        }

        return true;
    }

    void track(TreeNode* root, vector<int>& result) {
        if (!root) {
            return;
        }

        track(root->left, result);
        result.push_back(root->val);
        track(root->right, result);
    }
};
```

二叉搜索树中第K小的元素

```
class Solution {
public:
    int kthSmallest(TreeNode* root, int k) {    //迭代法中序遍历升序遍历节点
        stack<TreeNode*> st;
        TreeNode* cur = root;
        int count = 0;
        int result = 0;
        while (!st.empty() || cur) {
            while (cur) {   //深度优先搜索找到最小的元素
                st.push(cur);
                cur = cur->left;
            }

            cur = st.top(); //向上返回
            st.pop();
            count++;
            if (count == k) {
                result = cur->val;
            }

            cur = cur->right;
        }

        return result;
    }
};
```

二叉树的右视图

```
class Solution {
public:
    vector<int> rightSideView(TreeNode* root) {
        vector<int> result;
        if (!root) {
            return result;
        }

        queue<TreeNode*> que;
        que.push(root);
        while (!que.empty()) {
            int size = que.size();
            for (int i = 0; i < size; i++) {
                TreeNode* node = que.front();
                que.pop();
                if (node->left) que.push(node->left);
                if (node->right) que.push(node->right);
                if (i == size - 1) result.push_back(node->val);
            }
        }

        return result;
    }
};
```

二叉树展开为链表

```
class Solution {
public:
    void flatten(TreeNode* root) {
        if (!root) {
            return;
        }
        //后序遍历
        flatten(root->left);
        flatten(root->right);
        TreeNode* tmp = root->right;
        root->right = root->left;
        root->left = NULL;
        TreeNode* cur = root;
        while (cur->right) {        //找到展开后右子树的最右边节点
            cur = cur->right;
        }
        cur->right = tmp;       //拼接原来的右子树
    }
};
```

从前序与中序遍历序列构造二叉树

```
class Solution {
public:

    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        if (preorder.size() == 0 || inorder.size() == 0) {  //终止条件
            return NULL;
        }

        TreeNode* root = new TreeNode(preorder[0]); //前序遍历的第一个元素为根节点
        int i = 0;
        for (i = 0; i < inorder.size(); i++) {
            if (inorder[i] == root->val) {
```

```
                break;
            }
        }
        //vector构建方式
        vector<int> leftInorder(inorder.begin(), inorder.begin() + i);
        vector<int> rightInorder(inorder.begin() + i + 1, inorder.end());
        vector<int> leftPreOrder(preorder.begin() + 1, preorder.begin() + 1 + i);
        vector<int> rightPreOrder(preorder.begin() + 1 + i, preorder.end());


        root->left = buildTree(leftPreOrder, leftInorder);
        root->right = buildTree(rightPreOrder, rightInorder);
        return root;
    }
};
```

路径综合III

```
class Solution {
public:

    int dfs(TreeNode* root, int targetSum, long long sum) {
        if (!root) {
            return 0;
        }

        int count = 0;
        sum += root->val;
        if (sum == targetSum) {
            count++;
        }

        count += dfs(root->left, targetSum, sum);
        count += dfs(root->right, targetSum, sum);

        return count;
    }

    int pathSum(TreeNode* root, int targetSum) {
        if (!root) {
            return 0;
        }
        //层序遍历每一个节点
        int result = 0;
        queue<TreeNode*> que;
        que.push(root);
        while (!que.empty()) {
            int size = que.size();
            for (int i = 0; i < size; i++) {
                TreeNode* node = que.front();
                que.pop();
                result += dfs(node, targetSum, 0);  //以每一个节点为头进行深度优先搜索
                if (node->left) que.push(node->left);
                if (node->right) que.push(node->right);
            }
        }

        return result;

    }
};
```

```
//哈希表 前缀和 回溯
class Solution {
public:
    int pathSum(TreeNode* root, int targetSum) {
        unordered_map<long, int> umap;
        umap[0] = 1;
        int result = dfs(root, 0, targetSum, umap);
        return result;
    }

    int dfs(TreeNode* root, long curSum, int targetSum, unordered_map<long, int>&
umap) {
        if (!root) {
            return 0;
        }

        curSum += root->val;
        int preSum = umap[curSum - targetSum];

        umap[curSum]++;
        preSum += dfs(root->left, curSum, targetSum, umap);
        preSum += dfs(root->right, curSum, targetSum, umap);
        umap[curSum]--;

        return preSum;
    }
};
```

二叉树的最近公共祖先

```
class Solution {
public:
    //回溯查找二叉树根节点到目标节点的路径
    bool findPath(TreeNode* root, TreeNode* target, vector<TreeNode*>& path) {
        if (!root) return false;
        path.push_back(root);
        if (root == target) {
            return true;
        }

        if (findPath(root->left, target, path) || findPath(root->right, target,
path)) {
            return true;
        }

        path.pop_back();
        return false;
    }
    //根节点到p和q路径中最后一个相同的节点即为最近公共祖先
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        vector<TreeNode*> pathp;
        vector<TreeNode*> pathq;
        if (!findPath(root, p, pathp) || !findPath(root, q, pathq)) {
            return NULL;
        }

        TreeNode* result = NULL;
        for (int i = 0; i < min(pathp.size(), pathq.size()); i++) {
            if (pathp[i] == pathq[i]) {
                result = pathp[i];
```

```
            } else {
                break;
            }
        }

        return result;
    }
};
```

二叉树中的最大路径和

```
class Solution {
public:

    int dfs(TreeNode* root, int& res) {
        if (!root) {
            return 0;
        }

        int leftMax = max(dfs(root->left, res), 0);
        int rightMax = max(dfs(root->right, res), 0);
        res = max(res, leftMax + rightMax + root->val); //经过根结点
        return root->val + max(leftMax, rightMax);  //只选择一边返回给上一层
    }

    //最大路径和 后序遍历各节点为头节点，计算经过根结点和不经过根结点的最大值
    int maxPathSum(TreeNode* root) {
        int res = root->val;
        int result = dfs(root, res);
        return max(res, result);
    }
};
```

岛屿数量

```
class Solution {
public:
    int numIslands(vector<vector<char>>& grid) {
        int row = grid.size();
        int col = grid[0].size();
        if (row == 0) {
            return 0;
        }

        int nums = 0;

        vector<vector<bool>> visit(row, vector<bool>(col, false));
        for (int i = 0; i < row; i++) {
            for (int j = 0; j < col; j++) {
                if (grid[i][j] == '1' && visit[i][j] == false) {    //遍历每个岛屿进行
广度优先搜索
                    bfs(grid, i, j, visit);
                    nums++;
                }
            }
        }

        return nums;
    }
```

```
    void bfs(vector<vector<char>>& grid, int row, int col, vector<vector<bool>>&
visit) {
        queue<pair<int, int>> que;
        que.push({row, col});
        visit[row][col] = true;
        while (!que.empty()) {
            pair<int, int> tmp = que.front();
            que.pop();
            vector<vector<int>> dirct {{-1, 0}, {0, -1}, {1, 0}, {0, 1}};
            for (int i = 0; i < dirct.size(); i++) {
                int curRow = tmp.first + dirct[i][0];
                int curCol = tmp.second + dirct[i][1];
                if (curRow >= 0 && curRow < grid.size() && curCol >= 0
                && curCol < grid[0].size() && grid[curRow][curCol] == '1'
                && visit[curRow][curCol] == false) {
                    que.push({curRow, curCol});
                    visit[curRow][curCol] = true;
                }
            }
        }
    }
};
```

腐烂的橘子

```
class Solution {
public:
    int orangesRotting(vector<vector<int>>& grid) { //多源BFS
        queue<pair<int, int>> que;
        int rows = grid.size();
        int cols = grid[0].size();
        int fresh = 0;
        int times = 0;
        for (int i = 0; i < rows; i++) {
            for (int j = 0 ; j < cols; j++) {
                if (grid[i][j] == 2) {      //将所有腐烂的橘子入队
                    que.push({i, j});
                } else if (grid[i][j] == 1) {   //记录新鲜橘子的数量
                    fresh++;
                }
            }
        }

        vector<pair<int, int>> dirct{{-1, 0}, {0, -1}, {1, 0}, {0, 1}};
        while (!que.empty() && fresh > 0) {
            int size = que.size();
            for (int i = 0; i < size; i++) {
                pair<int, int> tmp = que.front();
                que.pop();
                for (int j = 0; j < dirct.size(); j++) {
                    int curRow = tmp.first + dirct[j].first;
                    int curCol = tmp.second + dirct[j].second;
                    if (curCol < 0 || curCol >= cols || curRow < 0
                        || curRow >= rows || grid[curRow][curCol] != 1) {
                            continue;
                        }

                    grid[curRow][curCol] = 2;   //新鲜橘子变为腐烂橘子
                    que.push({curRow, curCol});
                    fresh--;
                }
```

```
            }

            times++;
        }

        if (que.empty() && fresh != 0) {
            return -1;
        } else {
            return times;
        }
    }
};
```

课程表

```
class Solution {
public:
    bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {    //拓扑排序
        vector<vector<int>> graph(numCourses);
        vector<int> indegree(numCourses);

        for (auto& pre : prerequisites) {
            graph[pre[1]].push_back(pre[0]);    //先修课程指向后置课程
            indegree[pre[0]]++;     //课程的入度为要先修的课程数量
        }

        queue<int> que;
        for (int i = 0; i < numCourses; i++) {
            if (indegree[i] == 0) {
                que.push(i);
            }
        }

        int n = 0;
        while (!que.empty()) {
            int cur = que.front();
            que.pop();
            n++;

            for (int nei : graph[cur]) {
                indegree[nei]--;        //先修课程对应的后置课程入度-1
                if (indegree[nei] == 0) {
                    que.push(nei);
                }
            }
        }

        if (n == numCourses) {
            return true;
        } else {
            return false;
        }
    }
};
```

前缀树

```
class TrieNode {
public:
    unordered_map<char, TrieNode*> children;    //用哈希表保存树的结点
```

```
    bool isEnd;

    TrieNode() {
        children.clear();
        isEnd = false;
    }
};


class Trie {
private:
    TrieNode* root;

public:
    Trie() {
        root = new TrieNode();
    }

    void insert(string word) {
        TrieNode* node = root;
        for (int i = 0; i < word.size(); i++) {
            if (node->children.count(word[i]) == 0) {   //保存字符并构造下一个节点
                node->children[word[i]] = new TrieNode();
            }

            node = node->children[word[i]];
        }

        node->isEnd = true;
    }

    bool search(string word) {
        TrieNode* node = root;
        for (int i = 0; i < word.size(); i++) {
            if (node->children.count(word[i]) == 0) {
                return false;
            }

            node = node->children[word[i]];
        }

        return node->isEnd; //search要判断是否终止
    }

    bool startsWith(string prefix) {
        TrieNode* node = root;
        for (int i = 0; i < prefix.size(); i++) {
            if (node->children.count(prefix[i]) == 0) {
                return false;
            }

            node = node->children[prefix[i]];
        }

        return true;    //前缀查找只要存在即可
    }
};
```

全排列

```
class Solution {
```

```
public:

    vector<vector<int>> result;
    vector<int> path;

    void backTracking(vector<int>& nums, vector<bool> used) {
        if (path.size() == nums.size()) {
            result.push_back(path);
            return;
        }

        for (int i = 0; i < nums.size(); i++) {
            if (used[i] == true) {  //used数组去重
                continue;
            }

            path.push_back(nums[i]);
            used[i] = true;
            backTracking(nums, used);
            path.pop_back();
            used[i] = false;
        }
    }

    vector<vector<int>> permute(vector<int>& nums) {
        vector<bool> used(nums.size(), false);
        backTracking(nums, used);
        return result;
    }
};
```

子集

```
class Solution {
public:

    vector<vector<int>> result;
    vector<int> path;

    void backTracking(vector<int>& nums, int startIndex) {
        result.push_back(path);     //收集所有节点
        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;
    }
};
```

电话号码的字母组合

```
class Solution {
public:

    string nums[10] = {
        " ",        //0
```

```
        " ",
        "abc",
        "def",
        "ghi",
        "jkl",
        "mno",
        "pqrs",
        "tuv",
        "wxyz"
    };

    vector<string> result;
    string path;

    void backTracking(string digits, int startIndex) {
        if (path.size() == digits.size()) {
            result.push_back(path);
            return;
        }
        //遍历号码
        string digit = nums[digits[startIndex] - '0'];
        for (int i = 0; i < digit.size(); i++) {    //遍历字母
            path.push_back(digit[i]);
            backTracking(digits, startIndex + 1);   //下一个号码
            path.pop_back();
        }
    }

    vector<string> letterCombinations(string digits) {
        backTracking(digits, 0);
        return result;
    }
};
```

组合总和

```
class Solution {
public:

    vector<vector<int>> result;
    vector<int> path;

    void backTracking(vector<int> candidates, int target, int& sum, int startIndex)
{
        if (sum >= target) {
            if (sum == target) {
                result.push_back(path);
            }

            return;
        }

        for (int i = startIndex; i < candidates.size(); i++) {
            sum += candidates[i];
            path.push_back(candidates[i]);
            backTracking(candidates, target, sum, i);
            path.pop_back();
            sum -= candidates[i];
        }
    }
```

```
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        int sum = 0;
        backTracking(candidates, target, sum, 0);
        return result;
    }
};
```

括号生成

```
class Solution {
public:

    vector<string> result;
    string path;
    string s = "()";

    void backTracking(int n, int& leftCount, int& rightCount) {
        if (rightCount == n) {
            result.push_back(path);
            return;
        }

        for (int i = 0; i < s.size(); i++) {
            if (rightCount >= leftCount && s[i] == ')') {   //判断右括号是否能匹配
                continue;
            }

            if (leftCount > n) continue;    //判断左括号是否能匹配
            if (s[i] == '(') leftCount++;
            if (s[i] == ')') rightCount++;
            path.push_back(s[i]);
            backTracking(n, leftCount, rightCount);
            path.pop_back();
            if (s[i] == '(') leftCount--;
            if (s[i] == ')') rightCount--;
        }
    }

    vector<string> generateParenthesis(int n) {
        int leftCount = 0;
        int rightCount = 0;
        backTracking(n, leftCount, rightCount);
        return result;
    }
};
```

单词搜索

```
class Solution {
public:

    bool backTracking(vector<vector<char>>& board, string word, int row, int col,
int index, vector<vector<bool>>& visit) {
        if (index == word.size()) {
            return true;
        }

        if (row < 0 || col < 0 || row >= board.size() || col >= board[0].size() ||
board[row][col] != word[index] || visit[row][col]) {
            return false;
        }
```

```
        visit[row][col] = true; //去重
        bool res = false;
        res |= backTracking(board, word, row + 1, col, index + 1, visit);
        res |= backTracking(board, word, row - 1, col, index + 1, visit);
        res |= backTracking(board, word, row, col + 1, index + 1, visit);
        res |= backTracking(board, word, row, col - 1, index + 1, visit);
        visit[row][col] = false;

        return res;
    }


    bool exist(vector<vector<char>>& board, string word) {
        vector<vector<bool>> visit(board.size(), vector<bool>(board[0].size(),
false));
        for (int i = 0; i < board.size(); i++) {        //遍历棋盘所有位置当起点
            for (int j = 0; j < board[0].size(); j++) {
                if (backTracking(board, word, i, j, 0, visit)) {
                    return true;
                }
            }
        }

        return false;
    }
};
```

分割回文串

```
class Solution {
public:

    bool isValid(string s, int start, int end) {
        while (start <= end) {
            if (s[start] != s[end]) {
                return false;
            }

            start++;
            end--;
        }

        return true;
    }

    vector<vector<string>> result;
    vector<string> path;

    void backTracking(string s, int startIndex) {
        if (startIndex >= s.size()) {
            result.push_back(path);
            return;
        }

        for (int i = startIndex; i < s.size(); i++) {   //遍历字符串结束位置
            if (isValid(s, startIndex, i)) {
                path.push_back(s.substr(startIndex, i - startIndex + 1));
            } else {
                continue;
            }
```

```
            backTracking(s, i + 1); //选择下一个分割起始位置
            path.pop_back();
        }

    }

    vector<vector<string>> partition(string s) {
        backTracking(s, 0);
        return result;
    }
};
```

N皇后

```
class Solution {
public:

    vector<vector<string>> result;
    //判断是否能放置皇后
    bool isValid(vector<string>& board, int row, int col) {
        //同列
        for (int i = 0; i < row; i++) {
            if (board[i][col] == 'Q') {
                return false;
            }
        }
        //左上
        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
            if (board[i][j] =='Q') {
                return false;
            }
        }
        //右上
        for (int i = row - 1, j = col + 1; i >= 0 && j < board[0].size(); i--, j++)
{
            if (board[i][j] == 'Q') {
                return false;
            }
        }

        return true;
    }

    void backTracking(vector<string>& board, int startIndex) {
        if (startIndex >= board.size()) {
            result.push_back(board);
            return;
        }

        for (int col = 0; col < board[0].size(); col++) {
            if (isValid(board, startIndex, col)) {
                board[startIndex][col] = 'Q';   //放置皇后
                backTracking(board, startIndex + 1);
                board[startIndex][col] = '.';   //回溯
            }
        }
    }

    vector<vector<string>> solveNQueens(int n) {
        vector<string> board(n, string(n, '.'));
```

```
        backTracking(board, 0);

        return result;
    }
};
```

搜索插入位置

```
class Solution {
public:
    int searchInsert(vector<int>& nums, int target) {
        int left = 0;
        int right = nums.size() - 1;
        int mid = 0;
        while (left <= right) {
            mid = left + (right - left) / 2;
            if (nums[mid] > target) {
                right = mid - 1;
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else {
                return mid;
            }
        }
        //没有找到target，根据mid的值决定target的插入位置
        if (nums[mid] > target) {
            return mid;
        } else {
            return mid + 1;
        }
    }
};
```

搜索二维矩阵

```
class Solution {
public:
    bool searchMatrix(vector<vector<int>>& matrix, int target) {
        int rows = matrix.size();
        int cols = matrix[0].size();
        int left = 0;
        int right = rows * cols - 1;
        while (left <= right) {     //假设二维矩阵已经展开
            int mid = left + (right - left) / 2;
            int row = mid / cols;   //计算一维矩阵对应的行列索引
            int col = mid % cols;
            if (matrix[row][col] < target) {
                left = mid + 1;
            } else if (matrix[row][col] > target) {
                right = mid - 1;
            } else {
                return true;
            }
        }

        return false;
    }
};
```

在排序数组中查找元素的第一个和最后一个位置

```cpp
class Solution {
public:
    vector<int> searchRange(vector<int>& nums, int target) {
        int left = 0;
        int right = nums.size() - 1;
        int leftIndex = -1, rightIndex = -1;
        int mid = 0;
        while (left <= right) {
            mid = left + (right - left) / 2;
            if (nums[mid] > target) {
                right = mid - 1;
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else {
                leftIndex = mid;
                rightIndex = mid;
                break;
            }
        }

        if (leftIndex == -1) {
            return {leftIndex, rightIndex};
        }

        int tempRight = right;
        //找左侧第一个target值
        right = rightIndex - 1;
        while (left <= right) {
            mid = left + (right - left) / 2;
            if (nums[mid] > target) {
                right = mid - 1;
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else {
                leftIndex = mid;
                right = mid - 1;
            }
        }

        left = leftIndex + 1;
        right = tempRight;
        //找右侧最后一个target值
        while (left <= right) {
            mid = left + (right - left) / 2;
            if (nums[mid] > target) {
                right = mid - 1;
            } else if (nums[mid] < target) {
                left = mid + 1;
            } else {
                rightIndex = mid;
                left = mid + 1;
            }
        }

        return {leftIndex, rightIndex};
    }
};
```

搜索旋转排序数组

```cpp
class Solution {
```

```
public:
    int search(vector<int>& nums, int target) {
        int left = 0;
        int right = nums.size() - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if(nums[mid] == target) return mid;
            //mid在第一段单调递增区间
            if (nums[mid] >= nums[left]) {  //注意大于等于边界条件
                if (nums[left] <= target && target < nums[mid]) {
                    right = mid - 1;
                } else {
                    left = mid + 1;
                }
            } else {    //mid在第二段单调递增区间
                if (nums[mid] < target && target <= nums[right]) {
                    left = mid + 1;
                } else {
                    right = mid - 1;
                }
            }
        }

        return -1;
    }
};
```

寻找旋转排序数组中的最小值

```
class Solution {
public:
    int findMin(vector<int>& nums) {
        if (nums.size() == 1) {
            return nums[0];
        }
        int left = 0;
        int right = nums.size() - 1;
        while (left <= right) {
            int mid = left + (right - left) / 2;
            if (nums[left] <= nums[mid] && nums[mid] <= nums[right]) {
                return nums[left];  //找到单调递增区间
            }

            if (nums[mid] >= nums[left]) {
                left = mid + 1;
            } else if (nums[mid] <= nums[right]){
                right = mid;    //这里为mid
            }
        }

        return nums[left];
    }
};
```

寻找两个正序数组的中位数

```
class Solution {
public:
    double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
        if (nums1.size() > nums2.size()) {
            swap(nums1, nums2);
```

```
        }

        int m = nums1.size();
        int n = nums2.size();
        int left = 0;
        int right = m;
        while (left <= right) {
            int i = left + (right - left) / 2;  //分割线右边第一个元素的索引
            int j = (m + n + 1) / 2 - i;
            int left1 = (i == 0) ? INT_MIN : nums1[i - 1];
            int right1 = (i == m) ? INT_MAX : nums1[i];
            int left2 = (j == 0) ? INT_MIN : nums2[j - 1];
            int right2 = (j == n) ? INT_MAX : nums2[j];

            if (left1 <= right2 && left2 <= right1) {
                if ((m + n) % 2 == 1) { //总长度为奇数时多出的元素为中位数
                    return max(left1, left2);
                } else {    //总长度为偶数时计算中位数
                    return (max(left1, left2) + min(right1, right2)) / 2.0;
                }
            } else if (left1 > right2) {    //上面绳子靠右
                right = i - 1;      //往左移动
            } else if (left2 > right1) {    //上面绳子靠左
                left = i + 1;       //往右移动
            }
        }

        return -1;
    }
};
```

有效的括号

```
class Solution {
public:
    bool isValid(string s) {
        stack<char> st;
        st.push(s[0]);
        for (int i = 1; i < s.size(); i++) {
            if (s[i] == '(' || s[i] == '[' || s[i] == '{') {
                st.push(s[i]);
            } else if (s[i] == ')') {
                if (!st.empty() && st.top() == '(') {
                    st.pop();
                } else {
                    st.push(s[i]);
                }
            } else if (s[i] == ']') {
                if (!st.empty() && st.top() == '[') {
                    st.pop();
                } else {
                    st.push(s[i]);
                }
            } else if (s[i] == '}') {
                if (!st.empty() && st.top() == '{') {
                    st.pop();
                } else {
                    st.push(s[i]);
                }
            }
        }
```

```
        if (!st.empty()) {
            return false;
        } else {
            return true;
        }
    }
};
```

最小栈

```
class MinStack {
public:

    stack<int> st1;
    stack<int> st2;

    MinStack() {

    }

    void push(int val) {
        st1.push(val);
        if (!st2.empty()) {
            st2.push(min(st2.top(), val));
        } else {
            st2.push(val);
        }
    }

    void pop() {
        st1.pop();
        st2.pop();
    }

    int top() {
        return st1.top();
    }

    int getMin() {
        return st2.top();
    }
};
```

字符串解码

```
class Solution {
public:
    string decodeString(string s) {
        stack<char> st;
        for (int i = 0; i < s.size(); i++) {
            if (s[i] != ']') {
                st.push(s[i]);
            } else {    //遇到闭合的右括号
                string str; //字符串
                while (!st.empty() && st.top() != '[') {
                    str = st.top() + str;
                    st.pop();
                }

                st.pop(); //弹出左括号
```

```
                string num;
                while (!st.empty() && st.top() >= '0' && st.top() <= '9') {
                    num = st.top() + num;
                    st.pop();
                }

                int count = stoi(num);  //重复的次数
                string tmp = str;
                for (int i = 0; i < count - 1; i++) {
                    str += tmp;
                }

                for (int i = 0; i < str.size(); i++) {
                    st.push(str[i]);
                }
                // st.push(str);
            }
        }

        string result;
        while (!st.empty()) {
            result = st.top() + result;
            st.pop();
        }

        return result;
    }
};
```

每日温度

```
class Solution {
public:
    vector<int> dailyTemperatures(vector<int>& temperatures) {
        stack<int> st;
        vector<int> result(temperatures.size(), 0);
        st.push(0);
        for (int i = 1; i < temperatures.size(); i++) {
            if (temperatures[st.top()] >= temperatures[i]) {
                st.push(i);
            } else {
                while (!st.empty() && temperatures[st.top()] < temperatures[i]) {
                    result[st.top()] = i - st.top();
                    st.pop();
                }

                st.push(i);
            }
        }

        return result;
    }
};
```

柱状图中最大的矩形

```
class Solution {
public:
    int largestRectangleArea(vector<int>& heights) {
        //找出左右第一小的位置
        stack<int> st;
```

```
        heights.insert(heights.begin(), 0);
        heights.push_back(0);
        st.push(0);
        int result = 0;
        for (int i = 1; i < heights.size(); i++) {
            if (heights[i] >= heights[st.top()]) {
                st.push(i);
            } else {
                while (!st.empty() && heights[i] < heights[st.top()]) {
                    int height = heights[st.top()];
                    st.pop();
                    int width = i - st.top() - 1;
                    result = max(result, height * width);
                }

                st.push(i);
            }
        }

        return result;
    }
};
```

数组中的第K个最大元素

```
class Solution {
public:
    int findKthLargest(vector<int>& nums, int k) {
        priority_queue<int, vector<int>, greater<int>> minHeap;
        for (int num : nums) {
            if (minHeap.size() < k) {
                minHeap.push(num);
            } else if (num > minHeap.top()) {
                minHeap.pop();
                minHeap.push(num);
            }
        }

        return minHeap.top();
    }
};
```

前K个高频元素

```
class Solution {
public:
    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int, int> map;
        for (int i = 0; i < nums.size(); i++) {
            map[nums[i]]++;
        }

        vector<vector<int>> packet(nums.size() + 1);
        for (auto it = map.begin(); it != map.end(); it++) {
            packet[it->second].push_back(it->first);
        }

        vector<int> result;
        int count = 0;
        for (int i = packet.size() - 1; i >= 0; i--){
            if (packet[i].size() != 0) {
```

```
                for (int j = 0; j < packet[i].size(); j++){
                    result.push_back(packet[i][j]);
                    if (result.size() == k) {
                        return result;
                    }
                }
            }
        }

        return result;
    }
};
```

数据流的中位数

```
class MedianFinder {
public:

    priority_queue<int, vector<int>, less<int>> maxHeap;
    priority_queue<int, vector<int>, greater<int>> minHeap;

    MedianFinder() {

    }

    void addNum(int num) {
        maxHeap.push(num);
        if (maxHeap.size() - minHeap.size() > 1) {
            int tmp = maxHeap.top();
            maxHeap.pop();
            minHeap.push(tmp);
        }

        if (!maxHeap.empty() && !minHeap.empty() && maxHeap.top() > minHeap.top()) {
            int maxtmp = maxHeap.top();
            int mintmp = minHeap.top();
            maxHeap.pop();
            minHeap.pop();
            minHeap.push(maxtmp);
            maxHeap.push(mintmp);
        }
    }

    double findMedian() {
        if (maxHeap.size() == minHeap.size()) {
            return (maxHeap.top() + minHeap.top()) / 2.0;
        } else {
            return maxHeap.top() / 1.0;
        }
    }
};
```

买卖股票的最佳时机

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        vector<vector<int>> dp(prices.size(), vector<int>(2));
        dp[0][0] -= prices[0];        //持有股票
        dp[0][1] = 0;                //不持有股票
        for (int i = 1; i < prices.size(); i++) {
```

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

        return dp[prices.size() - 1][1];
    }
};

class Solution {
public:
    int maxProfit(vector<int>& prices) {
        int buy = prices[0];
        int result = 0;
        for (int i = 1; i < prices.size(); i++) {
            if (prices[i] < buy) {
                buy = prices[i];
            } else {
                result = max(result, prices[i] - buy);
            }
        }

        return result;
    }
};
```

跳跃游戏

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int target = nums.size() - 1;

        for (int i = nums.size() - 2; i >= 0; i--) {
            if (nums[i] + i >= target) target = i;
        }

        if (target == 0) return true;
        else return false;
    }
};
```

跳跃游戏 II

```
class Solution {
public:
    int jump(vector<int>& nums) {
        int left = 0;
        int right = 0;
        int count = 0;
        while (right < nums.size() - 1) {
            int fathest = 0;

            for (int i = left; i <= right; i++) {
                fathest = max(fathest, nums[i] + i);
            }

            left = right + 1;
            right = fathest;
            count += 1;
        }
```

```
        return count;
    }
};
```

划分字母区间

```
class Solution {
public:
    vector<int> partitionLabels(string s) {
        int right[26];
        vector<int> result;
        for (int i = 0; i < s.size(); i++) {
            right[s[i] - 'a'] = i;
        }

        int curmax = INT_MIN;
        int left = 0;
        for (int i = 0; i < s.size(); i++) {
            curmax = max(curmax, right[s[i] - 'a']);    //第二次遍历字符串
            if (i == curmax) {
                result.push_back(i - left + 1);
                left = i + 1;
            }
        }

        return result;
    }
};
```

爬楼梯

```
class Solution {
public:
    int climbStairs(int n) {
        if (n == 1) {
            return 1;
        }

        if (n == 2) {
            return 2;
        }

        vector<int> dp(n + 1);
        dp[1] = 1;
        dp[2] = 2;
        for (int i = 3; i < n + 1; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }

        return dp[n];
    }
};
```

杨辉三角

```
class Solution {
public:
    vector<vector<int>> generate(int numRows) {
        if (numRows == 1) {
            return {{1}};
        }
```

```
        if (numRows == 2) {
            return {{1}, {1, 1}};
        }

        vector<vector<int>> result;
        result.push_back({1});
        result.push_back({1, 1});
        for (int i = 2; i < numRows; i++){
            vector<int> lastRow = result[i - 1];
            vector<int> curRow;
            curRow.push_back(1);
            for (int i = 1; i < lastRow.size(); i++) {
                curRow.push_back(lastRow[i] + lastRow[i - 1]);
            }

            curRow.push_back(1);
            result.push_back(curRow);
        }

        return result;
    }
};
```

打家劫舍

```
class Solution {
public:
    int rob(vector<int>& nums) {
        if (nums.size() <= 1) {
            return nums[0];
        }
        vector<int> dp(nums.size());
        dp[0] = nums[0];
        dp[1] = max(nums[0], nums[1]);
        for (int i = 2; i < nums.size(); i++) {
            dp[i] = max(dp[i - 1], dp[i - 2] + nums[i]);
        }

        return dp[nums.size() - 1];
    }
};
```

完全平方数

```
class Solution {
public:
    int numSquares(int n) {
        vector<uint64_t> dp(n + 1, INT_MAX);
        dp[0] = 0;
        for (int i = 1; i * i <= n; i++) {
            for (int j = 0; j < n + 1; j++) {
                if (j >= i * i) {
                    dp[j] = min(dp[j], dp[j - i * i] + 1);
                }
            }
        }

        return dp[n];
    }
};
```

零钱兑换

```
class Solution {
public:
    int coinChange(vector<int>& coins, int amount) {
        vector<uint64_t> dp(amount + 1, INT_MAX);
        dp[0] = 0;
        for (int i = 0; i < coins.size(); i++) {
            for (int j = 0; j < amount + 1; j++) {
                if (j >= coins[i]) {
                    dp[j] = min(dp[j], dp[j - coins[i]] + 1);
                }
            }
        }

        if (dp[amount] == INT_MAX) {
            return -1;
        }
        return dp[amount];
    }
};
```

单词拆分

```
class Solution {
public:
    bool wordBreak(string s, vector<string>& wordDict) {
        unordered_set<string> unset(wordDict.begin(), wordDict.end());
        vector<bool> dp(s.size() + 1, false);
        dp[0] = true;
        for (int j = 1; j < s.size() + 1; j++) {    //先遍历背包
            for (int i = 0; i < j; i++) {
                string substr = s.substr(i, j - i);
                if (unset.find(substr) != unset.end() && dp[i]) {
                    dp[j] = true;
                }
            }
        }

        return dp[s.size()];
    }
};
```

最长递增子序列

```
class Solution {
public:
    int lengthOfLIS(vector<int>& nums) {
        if (nums.size() <= 1) return nums.size();
        vector<int> dp(nums.size(), 1);
        int result = 0;
        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);
            }

            if (dp[i] > result) result = dp[i];
        }

        return result;
```

```
    }
};
```

乘积最大子数组

```
class Solution {
public:
    int maxProduct(vector<int>& nums) {
        vector<int> maxdp(nums.size());
        vector<int> mindp(nums.size());
        maxdp[0] = nums[0];
        mindp[0] = nums[0];
        int result = nums[0];

        for (int i = 1; i < nums.size(); i++) {
            maxdp[i] = max({nums[i], maxdp[i - 1] * nums[i], nums[i] * mindp[i -
1]});
            mindp[i] = min({nums[i], maxdp[i - 1] * nums[i], nums[i] * mindp[i -
1]});
            result = max(result, maxdp[i]);
        }

        return result;
    }
};
```

分割等和子集

```
class Solution {
public:
    bool canPartition(vector<int>& nums) {
        int sum = 0;
        for (int i = 0 ; i < nums.size(); i++) {
            sum += nums[i];
        }

        if (sum % 2 == 1) {
            return false;
        }

        int target = sum / 2; //背包容量
        vector<int> dp(target + 1);
        for (int i = 0; i < nums.size(); i++) {
            for (int j = target; j >= nums[i]; j--) {
                dp[j] = max(dp[j], dp[j - nums[i]] + nums[i]);
            }
        }

        if (dp[target] == target) {
            return true;
        } else {
            return false;
        }
    }
};
```

最长有效括号

```
class Solution {
public:
    int longestValidParentheses(string s) {
```

```
        vector<int> dp(s.size());
        int result = 0;
        for (int i = 1; i < s.size(); i++) {
            if (s[i] == ')') {
                if (s[i - 1] == '(') {
                    dp[i] = i >= 2 ? dp[i - 2] + 2 : 2;
                } else if (s[i - 1] == ')') {
                    int m = i - dp[i - 1] - 1;
                    if (m >= 0 && s[m] == '(') {
                        dp[i] = m >= 1 ? 2 + dp[i - 1] + dp[m - 1] : 2 + dp[i - 1];
                    }
                }
            }
            result = max(result, dp[i]);
        }

        return result;
    }
};
```

不同路径

```
class Solution {
public:
    int uniquePaths(int m, int n) {
        vector<vector<int>> dp(m, vector<int>(n));
        dp[0][0] = 1;
        for (int i = 1; i < m; i++) {
            dp[i][0] = 1;
        }

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

最小路径和

```
class Solution {
public:
    int minPathSum(vector<vector<int>>& grid) {
        int m = grid.size();
        int 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] + grid[i][j], dp[i][j - 1] + grid[i]
[j]);
            }
        }

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

最长回文子串

```
class Solution {
public:
    string longestPalindrome(string s) {
        int n = s.size();
        if (n <= 1) return s;

        vector<vector<bool>> dp(n, vector<bool>(n, false));
        int start = 0, maxLen = 1;

        // 单个字符都是回文
        for (int i = 0; i < n; i++) {
            dp[i][i] = true;
        }

        // 枚举子串长度 len 从 2 到 n
        for (int len = 2; len <= n; len++) {
            for (int i = 0; i + len - 1 < n; i++) {
                int j = i + len - 1; // 子串结束位置
                if (s[i] == s[j]) {
                    if (len == 2 || dp[i + 1][j - 1]) {
                        dp[i][j] = true;
                        if (len > maxLen) {
                            maxLen = len;
                            start = i;
                        }
                    }
                }
            }
        }

        return s.substr(start, maxLen);
    }
};
```

最长公共子序列

```
class Solution {
public:
    int longestCommonSubsequence(string text1, string text2) {
        int m = text1.size();
        int n = text2.size();
        vector<vector<int>> dp(m + 1, vector<int>(n + 1));
        for (int i = 1; i <= m; i++) {
            for (int j = 1; j <= n; 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[m][n];
    }
};
```

编辑距离

```
class Solution {
public:
    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 i = 0; i <= word2.size(); i++) dp[0][i] = i;

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

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

只出现一次的数字

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {   //异或运算
        int result = 0;
        for (int i = 0; i < nums.size(); i++) {
            result ^= nums[i];
        }

        return result;
    }
};
```

多数元素

```
class Solution {
public:
    int majorityElement(vector<int>& nums) {    //摩尔计数
        int result = nums[0];
        int count = 1;
        for (int i = 1; i < nums.size(); i++) {
            if (nums[i] != result) {
                if (count) {
                    count--;
                } else {
                    count = 1;
                    result = nums[i];
                }
```

```
            } else {
                count++;
            }
        }

        return result;
    }
};
```

颜色分类

```
class Solution {
public:
    void sortColors(vector<int>& nums) {    //partition
        int i = 0;
        int left = 0;
        int right = nums.size() - 1;
        while (i <= right) {
            if (nums[i] == 2) { // i=2 与right交换
                swap(nums[i], nums[right]);
                right--;
            } else if (nums[i] == 0) {  // i = 0 与left交换
                swap(nums[i], nums[left]);
                left++;
                i++;
            } else if (nums[i] == 1) {  // i = 1 不交换
                i++;
            }
        }
    }
};
```

下一个排列

```
class Solution {
public:
    void nextPermutation(vector<int>& nums) {
        int i;
        for (i = nums.size() - 1; i >= 0; i--) {
            if (i + 1 < nums.size() && nums[i] < nums[i + 1]) { //在i后查找是否有比i更
大的元素
                for (int j = nums.size() - 1; j > i; j--) {
                    if (nums[j] > nums[i]) {
                        swap(nums[j], nums[i]);
                        break;
                    }
                }

                break;
            }
        }

        int right = nums.size() - 1;    //交换i后的所有元素，进行升序排列
        int left = i + 1;
        while (left <= right) {
            swap(nums[left], nums[right]);
            left++;
            right--;
        }
    }
};
```

寻找重复数

```cpp
class Solution {
public:
    int findDuplicate(vector<int>& nums) {  //找环的入口
        int slow = 0;
        int fast = 0;
        while (true) {
            slow = nums[slow];
            fast = nums[nums[fast]];
            if (slow == fast) {
                break;
            }
        }

        fast = 0;
        while (true) {
            fast = nums[fast];
            slow = nums[slow];
            if (fast == slow) {
                return fast;
            }
        }

        return 0;
    }
};
```