☆回溯

☆贪心

☆目录

> 二分查找：704, 33
>
> 双指针：27, 26, 977, O-5, 209, 15, 18
>
> 链表：203, 707, 206, 24, 19, 160, 142
>
> 哈希表：242, 349, 202, 1, 454, 383
>
> 字符串：344, 541, 151, 58, 28, 459
>
> 栈与队列：232, 225, 20, 1047, 150, 347, 239
>
> 排序：O-51, 315, 215
>
> 二叉树：144, 94, 145, 102, 226, 101, 100, 104, 111, 222, 110, 257, 404, 513,
>
> 112, 113, 106, 105, 654, 617, 700, 98, 530, 501, 236, 235, 701, 450, 669, 108, 538
>
> 回溯：124, 77, 216, 17, 39, 40, 131, 93, 78, 90, 491, 46, 47, 332, 51, 37
>
> 贪心：455, 376, 53, 122, 55, 45, 1005, 134, 135, 860, 406, 452, 435, 763, 56, 738, 968
>
> 动态规划：509, 70, 62, 63, 343, 96, 416, 1049, 494
>
> 其他：59

☆二分查找

模板 #704

```
int search(vector<int>& nums, int target){
    int n = (int)nums.size();
    if (!n)  return -1;
    if (n == 1)  return nums[0] == target ? 0 : -1;

    int left = 0, right = n - 1;
    while(left <= right){
        int middle = left + (right - left) / 2;
        if(target < nums[middle]) right = middle - 1;
        else if(target > nums[middle]) left = middle + 1;
        else return middle;
    }
    return -1;
}
```

搜索旋转排序数组 #33

```
class Solution {
public:
    int search(vector<int>& nums, int target) {
        int n = nums.size();
        if(!n) return -1;
        if(n == 1) return nums[0]==target ? 0:-1;
        int left = 0, right = n - 1;
        while(left <= right){
            int mid = left + (right - left)/2;
            if(nums[mid] == target) return mid;
            if(nums[left] <= nums[mid]){
                if(nums[left] <= target && target <= nums[mid]) right = mid - 1;
                else left = mid + 1;
            }else{
                if(nums[mid] <= target && target <= nums[right]) left = mid + 1;
                else right = mid - 1;
            }
        }
    return -1;
    }
};
```

☆双指针

移除元素 #27

```
// 时间复杂度：O(n)
// 空间复杂度：O(1)
class Solution {
public:
    int removeElement(vector<int>& nums, int val) {
        int slowIndex = 0;
        for (int fastIndex = 0; fastIndex < nums.size(); fastIndex++) {
            if (val != nums[fastIndex]) {
                nums[slowIndex++] = nums[fastIndex];
            }
        }
        return slowIndex;
    }
};
```

删除有序数组重复项 #26

```
class Solution {
public:
    int removeDuplicates(vector<int>& nums) {
        int slow = 0, n = nums.size();
        if(!n) return -1;
        if(n == 1) return 1;

        for(int fast = 0; fast < n; fast++) {
            if(nums[slow] != nums[fast]) {
                nums[slow++] = nums[fast];
            }
        }
        return slow;
    }
};
```

有序数组的平方 #977

```
class Solution {
public:
    vector<int> sortedSquares(vector<int>& nums) {
        int n = nums.size();
        vector<int> res(n, 0);
        int k = n - 1;

        for(int i = 0, j = n - 1; i <= j; ) {
            if(nums[i] * nums[i] < nums[j] * nums[j]) {
                res[k--] = nums[j] * nums[j];
                j--;
            }else {
                res[k--] = nums[i] * nums[i];
                i++;
            }
        }
        return res;
    }
};
```

替换空格 #O-5

```
class Solution {
public:
    string replaceSpace(string s) {
        int oldsize = s.size(), cnt = 0;
        for(int i = 0; i < oldsize; i++) {
            if(s[i] == ' ') cnt++;
        }
        s.resize(oldsize + cnt * 2);
        int newsize = s.size();
        for(int i = oldsize - 1, j = newsize - 1; i < j ; i--, j--) {
            if(s[i] != ' ') s[j] = s[i];
            else{
                s[j] = '0';
                s[j - 1] = '2';
                s[j - 2] = '%';
                j = j - 2;
            }
        }
        return s;
    }
};
```

长度最小的子数组 #209

滑动窗口

```
class Solution {
public:
    int minSubArrayLen(int target, vector<int>& nums) {
        int result = INT32_MAX;
        int sum = 0, i = 0;
        int n = nums.size();
        if (n < 1) return 0;
        for(int j = 0; j < n; j++) {
            sum += nums[j];
            while(sum >= target) {
                int length = j - i + 1;
                result = result > length ? length : result;
                sum -= nums[i++];
            }
        }
        return result == INT32_MAX ? 0 : result;
    }
};
```

三数之和 #15

```
class Solution {
public:
    vector<vector<int>> threeSum(vector<int>& nums) {
        int n = nums.size();
        if(n < 3) return {};
        sort(nums.begin(), nums.end());
        vector<vector<int>> res;
        for(int i = 0; i < n; i++) {
            if(nums[i] > 0) return res;
            if(i > 0 && nums[i] == nums[i - 1]) continue;
            int left = i + 1, right = n - 1;
            while(left < right) {
                if(nums[i] + nums[left] + nums[right] > 0) {
                    right--;
                    while(left < right && nums[right] == nums[right + 1]) right-
-;
                } else if(nums[i] + nums[left] + nums[right] < 0) {
                    left++;
                    while(left <  right && nums[left] == nums[left - 1]) left++;
                } else {
                    res.push_back(vector<int>{nums[i], nums[left],
nums[right]});
                    right--;
                    left++;
                    while(left < right && nums[right] == nums[right + 1]) right-
-;
                    while(left <  right && nums[left] == nums[left - 1]) left++;
                }
            }
        }
        return res;
    }
};
```

四数之和 #18

```
class Solution {
public:
    vector<vector<int>> fourSum(vector<int>& nums, int target) {
        int n = nums.size();
        if(n < 4) return {};
        vector<vector<int>> res;
        sort(nums.begin(), nums.end());

        for(int k = 0; k < n; k++) {
            if(k > 0 && nums[k] == nums[k - 1]) continue;
            for(int i = k + 1; i < n; i++) {   // i = k + 1
                if(i > k + 1 && nums[i] == nums[i - 1]) continue;
                int left = i + 1;
                int right = n - 1;
                while(left < right) {
                    if(nums[k] + nums[i] - target > - nums[left] - nums[right])
{
                        right--;
                        while(left < right && nums[right] == nums[right + 1])
right--;
                    } else if(nums[k] + nums[i] - target < - nums[left] -
nums[right]) {
                        left++;
                        while(left < right && nums[left] == nums[left - 1])
left++;
                    } else {
                        res.push_back(vector<int>{nums[k], nums[i], nums[left],
nums[right]});
                        left++;
                        right--;
                        while(left < right && nums[right] == nums[right + 1])
right--;
                        while(left < right && nums[left] == nums[left - 1])
left++;
                    }
                }
            }
        }
        return res;
    }
};
```

☆链表

定义

```
struct ListNode {
    int val;
    ListNode *next;
    ListNode(int x) : val(x), next(NULL) {}
};
```

删除元素 #203

```
class Solution {
public:
    ListNode* removeElements(ListNode* head, int val) {
        ListNode* dummyHead = new ListNode(0);
        dummyHead->next = head;
        ListNode* cur = dummyHead;
        while(cur->next != nullptr) {
            if(cur->next->val == val) {
                ListNode* tmp = cur->next;
                cur->next = cur->next->next;
                delete tmp;
            }else {
                cur = cur->next;
            }
        }
        head = dummyHead->next;
        delete dummyHead;
        return head;
    }
};
```

插入元素 #707

```
void addAtIndex(int index, int val) {
    if(index > _size) return;

    ListNode* newNode = new ListNode(val);
    ListNode* cur = _dummyHead;
    while(index--) {
        cur = cur->next;
    }
    newNode->next = cur->next;
    cur->next = newNode;
}
```

翻转链表 #206

```cpp
class Solution {
public:
    ListNode* reverseList(ListNode* head) {
        ListNode* tmp;
        ListNode* cur = head;
        ListNode* pre = nullptr;
        while(cur) {
            tmp = cur->next;
            cur->next = pre;
            pre = cur;
            cur = tmp;
        }
        return pre;
    }
};
```

两两交换链表中的节点 #24

```cpp
class Solution {
public:
    ListNode* swapPairs(ListNode* head) {
        ListNode* dummyHead = new ListNode(0);
        dummyHead->next = head;
        ListNode* cur = dummyHead;
        while(cur->next != nullptr && cur->next->next != nullptr) {
            ListNode* tmp1 = cur->next;
            ListNode* tmp2 = cur->next->next->next;

            cur->next = cur->next->next;
            cur->next->next = tmp1;
            cur->next->next->next = tmp2;
            cur = cur->next->next;
        }
        return dummyHead->next;
    }
};
```

删除链表倒数第N个数 #19

```cpp
class Solution {
public:
    ListNode* removeNthFromEnd(ListNode* head, int n) {
        ListNode* dummyHead = new ListNode(0);
        dummyHead->next = head;
        ListNode* fast = dummyHead;
        ListNode* slow = dummyHead;

        while(n-- && fast != nullptr) {
            fast = fast->next;
        }
        fast = fast->next;
        while(fast != nullptr) {
            fast = fast->next;
            slow = slow->next;
        }
        ListNode* tmp = slow->next;
        slow->next = slow->next->next;
        delete tmp;
        return dummyHead->next;
    }
};
```

链表相交 #160

```
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        int lenA = 0, lenB = 0;
        ListNode* curA = headA;
        ListNode* curB = headB;

        while(curA != nullptr) {
            lenA++;
            curA = curA->next;
        }
        while(curB != nullptr) {
            lenB++;
            curB = curB->next;
        }

        curA = headA;
        curB = headB;
        if(lenB > lenA) {
            swap(lenA, lenB);
            swap(curA, curB);
        }
        int gap = lenA - lenB;
        while(gap--) {
            curA = curA->next;
        }
        while(curA != nullptr) {
            if(curA == curB) return curA;
            curA = curA->next;
            curB = curB->next;
        }
        return nullptr;
    }
};
```

环形链表II #142

```
class Solution {
public:
    ListNode *detectCycle(ListNode *head) {
        ListNode* fast = head;
        ListNode* slow = head;
        while(fast != nullptr && fast->next != nullptr) {
            fast = fast->next->next;
            slow = slow->next;
            if(fast == slow) {
                ListNode* index1 = fast;
                ListNode* index2 = head;
                while(index1 != index2) {
                    index1 = index1->next;
                    index2 = index2->next;
                }
                return index1;
            }
        }
        return nullptr;
    }
};
```

☆哈希表

字母异位词 #242

```
class Solution {
public:
    bool isAnagram(string s, string t) {
        int record[26] = {0};
        int i = 0;
        for(i = 0; i < s.size(); i++) {
            record[s[i]-'a']++;
        }
        for(i = 0; i < t.size(); i++) {
            record[t[i]-'a']--;
        }
        for(i = 0; i < 26; i++) {
            if(record[i] != 0) return false;
        }
        return true;
    }
};
```

两个数组的交集 #349

```
class Solution {
public:
    vector<int> intersection(vector<int>& nums1, vector<int>& nums2) {
        unordered_set<int> res;
        unordered_set<int> nums1_set(nums1.begin(), nums1.end());

        for(int num : nums2) {
            if(nums1_set.find(num) != nums1_set.end()) {
                res.insert(num);
            }
        }
        return vector<int>(res.begin(), res.end());
    }
};
```

快乐数 #202

```
class Solution {
public:
    int getSum(int n) {
        int sum = 0;
        while(n) {
            sum += (n % 10) * (n % 10);
            n = n / 10;
        }
        return sum;
    }
    bool isHappy(int n) {
        unordered_set<int> set;
        while(1) {
            int sum = getSum(n);
            if(sum == 1) return true;
            if(set.find(sum) != set.end()) return false;
            set.insert(sum);
            n = sum;
        }
    }
};
```

两数之和 #1

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

四数相加II #454

```
class Solution {
public:
    int fourSumCount(vector<int>& nums1, vector<int>& nums2, vector<int>& nums3, vector<int>& nums4) {
        unordered_map <int, int> umap;
        for(int a : nums1) {
            for(int b : nums2) {
                umap[a + b]++;
            }
        }
        int cnt = 0;
        for(int c : nums3) {
            for(int d : nums4) {
                if(umap.find(0 - (c + d)) != umap.end()) {
                    cnt += umap[0 - (c + d)];
                }
            }
        }
        return cnt;
    }
};
```

赎金信 #383

☆字符串

反转字符串 #344

```
// I.
class Solution {
public:
    void reverseString(vector<char>& s) {
        for(int i = 0, j = s.size() - 1; i < s.size() /  2; i++, j--) {
            swap(s[i], s[j]);
        }
    }
};
```

反转字符串 #541

```
// II.
class Solution {
public:
    void reverse(string& s, int start, int end) {
        for(int i = start, j = end - 1; i < start + (end - start) / 2; i++, j--) {  // i < start +
            swap(s[i], s[j]);
        }
     }
    string reverseStr(string s, int k) {
        int n = s.size();
        for(int i = 0; i < n; i += 2 * k) {
            if(i + k < n) {
                reverse(s, i, i + k);
            } else {
                reverse(s, i, n);
            }
        }
        return s;
    }
};
```

翻转字符串里的单词 #151

左旋转字符串 #58

```
// 1.
class Solution {
public:
    string reverseLeftWords(string s, int n) {
        reverse(s.begin(), s.begin() + n);
        reverse(s.begin() + n, s.end());
        reverse(s.begin(), s.end());
        return s;
    }
};
// 2.
class Solution {
public:
    string reverseLeftWords(string s, int n) {
        string res = string(s);
        int j = 0;
        for(int i = n; i < s.size(); i++) {
            res[j++] = s[i];
        }
        for(int i = 0; i < n; i++) {
            res[j++] = s[i];
        }
        return res;
    }
};
```

KMP #28

```
class Solution {
public:
    void getNext(int* next, string& s) {
        int j = 0;
        next[0] = 0;
        for(int i = 1; i < s.size(); i++) { // i = 1 !
            while(j > 0 && s[i] != s[j]) {
                j = next[j - 1];
            }
            if(s[i] == s[j]) {
                j++;
            }
            next[i] = j;
        }
    }
    int strStr(string haystack, string needle) {
        int n = needle.size();
        if(!n) return 0;
        int next[n];
        getNext(next, needle);

        int j = 0;
        for(int i = 0; i < haystack.size(); i++) { // i = 0
            while(j > 0 && haystack[i] != needle[j]) {
                j = next[j - 1];
            }
            if(haystack[i] == needle[j]) {
                j++;
            }
            if(j == n) {
                return (i - j + 1);
            }
        }
        return -1;
    }
};
```

重复的子字符串 #459

暴力

```
class Solution {
public:
    bool repeatedSubstringPattern(string s) {
        int n = s.size();
        for(int i = 1; i <= n / 2; i++) {  // i = 1
            if(n % i == 0) {
                bool match = true;  // before for()
                for(int j = i; j < n; j++) {
                    if(s[j] != s[j - i]) {
                        match = false;
                        break;  // break
                    }
                }
                if(match) return true;
            }
        }
        return false;
    }
};
```

KMP

```
bool repeatedSubstringPattern (string s) {
    if (s.size() == 0) {
        return false;
    }
    int next[s.size()];
    getNext(next, s);
    int len = s.size();
    if (next[len - 1] != 0 && len % (len - (next[len - 1] )) == 0) {
        return true;
    }
    return false;
}
```

☆栈与队列

栈实现队列 #232

```cpp
class MyQueue {
public:
    stack<int> stack_in;
    stack<int> stack_out;
    MyQueue() {

    }

    void push(int x) {
        stack_in.push(x);
    }

    int pop() {
        if(stack_out.empty()) {
            while(!stack_in.empty()) {
                stack_out.push(stack_in.top());
                stack_in.pop();
            }
        }
        int res = stack_out.top();
        stack_out.pop();
        return res;
    }

    int peek() {
        int res = this->pop();
        stack_out.push(res);
        return res;
    }

    bool empty() {
        return stack_in.empty() && stack_out.empty();
    }
};
```

队列实现栈 #225

```cpp
class MyStack {
public:
    queue<int> q;
    MyStack() {

    }

    void push(int x) {
        q.push(x);
    }

    int pop() {
        int size = q.size();
        while(--size) {
            q.push(q.front());
            q.pop();
        }
        int res = q.front();
        q.pop();
        return res;
    }

    int top() {
        return q.back();
    }

    bool empty() {
        return q.empty();
    }
};
```

有效的括号 #20

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

删除字符串相邻重复项 #1047

```
// 1. stack
class Solution {
public:
    string removeDuplicates(string s) {
        stack<char> st;
        for(char ch : s) {
            if(st.empty() || ch != st.top()) { // st.empty() is necessary
                st.push(ch);
            } else {
                st.pop();
            }
        }

        string res = "";
        while(!st.empty()) {
            res += st.top();  // string +=
            st.pop();
        }
        reverse(res.begin(),res.end());
        return res;
    }
};
// 2. string
```

```
class Solution {
public:
    string removeDuplicates(string s) {
        string res = "";
        for(char ch : s) {
            if(res.empty() || ch != res.back()) {
                res.push_back(ch);
            } else {
                res.pop_back();
            }
        }
        return res;

    }
};
```

逆波兰 #150

```
class Solution {
public:
    int evalRPN(vector<string>& tokens) {
        stack<int> st;
        for(int i = 0; i < tokens.size(); i++) {
            if(tokens[i] == "+" || tokens[i] == "-" || tokens[i] == "*" ||
tokens[i] == "/") {
                int num1 = st.top();
                st.pop();
                int num2 = st.top();
                st.pop();
                if(tokens[i] == "+") st.push(num2 + num1);
                else if(tokens[i] == "-") st.push(num2 - num1);
                else if(tokens[i] == "*") st.push(num2 * num1);
                else if(tokens[i] == "/") st.push(num2 / num1);
            } else {
                st.push(stoi(tokens[i]));
            }
        }
        return st.top();
    }
};
```

前k个高频元素 #347

```
class Solution {
public:
    struct cmp{  // operator
        bool operator()(pair<int, int>& p1, pair<int, int>& p2) {
            return p1.second > p2.second;
        }
    };

    vector<int> topKFrequent(vector<int>& nums, int k) {
        unordered_map<int, int> map;
        for(int num : nums) {
            map[num]++;
        }
        priority_queue<pair<int, int>, vector<pair<int, int>>, cmp> pri_que;
        for(unordered_map<int, int>::iterator it = map.begin(); it != map.end();
it++) {  // spell:iterator,begin
            pri_que.push(*it);
            if(pri_que.size() > k) {
                pri_que.pop();
            }
        }

        vector<int> res(k);  // (k)
        for(int i = k - 1; i >= 0; i--) {
            res[i] = pri_que.top().first;  // 1 top not front  2 .first
            pri_que.pop();  // not forget pop
        }
        return res;
    }
};
```

滑动窗口最大值 #239

单调队列

```
// 1.
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        deque<int> q;  //双向列表，保存下标，实现单调(递减)队列
        for(int i = 0; i < k; i++) {
            while(!q.empty() && nums[i] >= nums[q.back()]) {
                q.pop_back();
            }
            q.push_back(i);
        }

        vector<int> res = {nums[q.front()]};  // initiate
        for(int i = k; i < nums.size(); i++) {
            while(!q.empty() && nums[i] >= nums[q.back()]) {  // 1.保持单调递减
                q.pop_back();
            }
            q.push_back(i);

            while(q.front() <= i - k) {  // 2.保证队首元素下标在窗口内
                q.pop_front();
            }
            res.push_back(nums[q.front()]); // 队首永远是最大元素的下标
        }
        return res;
    }
};
// 2.
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        deque<int> q;  //双向列表，保存下标，实现单调(递减)队列
        vector<int> res;  // initiate
        for(int i = 0; i < nums.size(); i++) {
            while(!q.empty() && nums[i] >= nums[q.back()]) {  // 1.保持单调递减
                q.pop_back();
            }
            q.push_back(i);

            while(q.front() <= i - k) {  // 2.保证队首元素下标在窗口内
                q.pop_front();
            }
            if(i >= k - 1) {
                res.push_back(nums[q.front()]); // 队首永远是最大元素的下标
            }
        }
        return res;
    }
};
```

☆排序

归并排序

```
void merge_sort(int q[], int l, int r){
    if(l >= r) return;
    int mid = l + r >> 1;
    merge_sort(q, l, mid);    // [l,mid] && [mid+1,r]
    merge_sort(q, mid+1, r);  // 执行完后，说明两个区间已经排好序

    int k = 0, i = l, j = mid + 1;
    while(i <= mid && j <= r){
        if(q[i] <= q[j]) temp[k ++] = q[i ++];
        else temp[k ++] = q[j ++];
    }
    while(i <= mid) temp[k ++] = q[i ++];  // 别忘了扫尾
    while(j <= r) temp[k ++] = q[j ++];

    for(int i = l, j = 0; i <= r; i ++, j ++) q[i] = temp[j];  // 复制回原数组
}
```

逆序对 #O-51

```
class Solution {
public:
    vector<int> tmp;
    int merge_sort_rp(vector<int>& nums, vector<int>& tmp, int l, int r) {
        if(l >= r) return 0;
        int mid = l + r >> 1;
        int res = merge_sort_rp(nums, tmp, l, mid) + merge_sort_rp(nums, tmp,
mid + 1, r);

        int i = l, j = mid  + 1, k = 0;
        while(i <= mid && j <= r) {
            if(nums[i] <= nums[j]) {
                tmp[k++] = nums[i++];
            } else {
                tmp[k++] = nums[j++];
                res += mid - i + 1;
            }
        }
        while(i <= mid) tmp[k++] = nums[i++];
        while(j <= r) tmp[k++] = nums[j++];
        for(int i = l, j = 0; i <= r; i++, j++) {
            nums[i] = tmp[j];
        }
        return res;
    }
    int reversePairs(vector<int>& nums) {
        int n = nums.size();
        vector<int> tmp(n);
        return merge_sort_rp(nums, tmp, 0, nums.size() - 1);
    }
};
```

右边逆序数 #315

```
class Solution {
public:
    vector<pair<int, int>> tmp;
    vector<int> res;
    vector<int> countSmaller(vector<int>& nums) {
        int n = nums.size();
        vector<pair<int, int>> nums_index;
        for(int i = 0; i < n; i++) {
            nums_index.push_back(pair<int, int>(nums[i], i));
        }

        tmp = vector<pair<int, int>>(n);
        res = vector<int>(n, 0);

        merge_sort(nums_index, 0, n - 1);

        return res;
    }
    void merge_sort(vector<pair<int, int>>& nums_index, int l, int r) {
        if(l >= r) return;

        int mid = l + r >> 1;
        merge_sort(nums_index, l, mid);
        merge_sort(nums_index, mid + 1, r);
        merge(nums_index, l, mid, r);
    }
    void merge(vector<pair<int, int>>& nums_index, int l, int mid, int r){
        int i = l, j = mid + 1, k = l;
        while(i <= mid && j <= r) {
            if(nums_index[i].first <= nums_index[j].first) {
                res[nums_index[i].second] += j - mid -1;
                tmp[k++] = nums_index[i++];
            } else {
                tmp[k++] = nums_index[j++];
            }
        }
        while(i <= mid) {
            res[nums_index[i].second] += j - mid -1;
            tmp[k++] = nums_index[i++];
        }
        while(j <= r) {
            tmp[k++] = nums_index[j++];
        }
        for(i = l; i <= r; i++) {
            nums_index[i] = tmp[i];
        }
    }
};
```

快速排序

```
void quick_sort(int q[], int l, int r) {
    if (l >= r) return;//提前判断
    int x = q[l + r >> 1];
    int i = l - 1, j = r + 1;  // l-1, r+1
    while (i < j) {  //都是<
        while (q[++i] < x);
        while (q[--j] > x);
        if (i < j) swap(q[i], q[j]);
    }
    quick_sort(q, l, j);  // l,j
    quick_sort(q, j + 1, r);  // j+1,r
}
```

第K个最大的数 #215

```
class Solution {
public:
    int quick_sort(vector<int>& nums, int l, int r, int k) {
        if(l > r) return -1;
        if(l == r) return nums[l];

        int x = nums[l + r >> 1];
        int i = l - 1, j = r + 1;
        while(i < j) {
            while(nums[++i] < x);
            while(nums[--j] > x);
            if(i < j) swap(nums[i], nums[j]);
        }

        // int left = j - l + 1;  // 第K个最小的数
        // if(k <= left) return quick_sort(nums, l, j, k);
        // else return quick_sort(nums, j + 1, r, k - left);
        int right = r - j;
        if(k <= right) return quick_sort(nums, j + 1, r, k);  // [l,j] && [j+1,r] 长度分别为 <j-i+1>和<r-j>
        else return quick_sort(nums, l, j, k - right);  // k-right
    }
    int findKthLargest(vector<int>& nums, int k) {
        return quick_sort(nums, 0, nums.size() - 1, k);
    }
};
```

☆二叉树

定义

```
struct TreeNode {
    int val;
    TreeNode *left;
    TreeNode *right;
    TreeNode(int x) : val(x), left(NULL), right(NULL) {}
};
```

深度遍历 #144,94,145

前中后序 #144 #94 #145

```
class Solution {
public:
    void traverse(TreeNode *cur, vector<int>& vec) {
        if(cur == nullptr) return;
        vec.push_back(cur->val);
        traverse(cur->left, vec);
        traverse(cur->right, vec);
    }

    vector<int> preorderTraversal(TreeNode* root) {
        vector<int> res;
        traverse(root, res);
        return res;
    }
};

void traverse(TreeNode *cur, vector<int>& vec) {
    if(cur == nullptr) return;
    traverse(cur->left, vec);
    vec.push_back(cur->val);
    traverse(cur->right, vec);
}
void traverse(TreeNode *cur, vector<int>& vec) {
    if(cur == nullptr) return;
    traverse(cur->left, vec);
    traverse(cur->right, vec);
    vec.push_back(cur->val);
}
```

☆迭代遍历（栈）

层序遍历 #102

```
class Solution {
public:
    vector<vector<int>> levelOrder(TreeNode* root) {
        vector<vector<int>> res;
        if(!root) return res;

        queue<TreeNode*> que;
        que.push(root);

        while(!que.empty()) {
            vector<int> vec;
            int size = que.size();
            // 一定要提前计算que.size(),因为会变
            // 也不要写 while(que.empty()),也是因为que.size()会变
            for(int i = 0; i < size; i++) {
                TreeNode* node = que.front();
                vec.push_back(node->val);
                que.pop();

                if(node->left) que.push(node->left);
                if(node->right) que.push(node->right);
                // delete tmp; // 不可以写删除，否则会段错误
            }
            res.push_back(vec);
        }
        return res;
    }
};
```

翻转二叉树 #226

```
class Solution {
public:
    TreeNode* invertTree(TreeNode* root) {
        if(!root) return nullptr;
        // if(root->left == nullptr && root->right == nullptr) return root;

        swap(root->left, root->right);     //前序遍历
        invertTree(root->left);            //中序遍历不可以，会翻转两次
        invertTree(root->right);
        // swap(root->left, root->right);  //后序遍历

        return root;
    }
};
```

对称二叉树 #101

```
class Solution {
public:
    bool compare_sym(TreeNode* left,TreeNode* right) {
        if(!left && !right) return true;
        else if((!left && right) || (left && !right)) return false;
        else if(left->val != right->val) return false;

        // 左右节点val相等，传入“左左，右右”和“左右，右左”两对节点，递归遍历外侧和内侧的树是否对称
        return compare_sym(left->left, right->right) && compare_sym(left->right, right->left);
    }
    bool isSymmetric(TreeNode* root) {
        if(!root) return true;
        return compare_sym(root->left, root->right);
    }
};
```

相同的树 #100

```
class Solution {
public:
    bool isSameTree(TreeNode* p, TreeNode* q) {
        if(!p && !q) return true;
        else if(!p && q || p && !q) return false;
        else if(p->val != q->val) return false;
        return isSameTree(p->left, q->left) && isSameTree(p->right, q->right);
    }
};
```

二叉树的最大深度 #104

```
class Solution {
public:
    int maxDepth(TreeNode* root) {
        if(!root) return 0;
        // if(!root->left && !root->right) return 1;
        return 1 + max(maxDepth(root->left), maxDepth(root->right));
    }
};
// n叉树最大深度
class Solution {
public:
    int maxDepth(Node* root) {
        if(!root) return 0;
        int depth = 0;
        for(Node* node : root->children) {
            int tmp = maxDepth(node);
            depth = depth > tmp ? depth : tmp;
        }
        return depth + 1;
    }
};
```

二叉树的最小深度 #111

```
class Solution {
public:
    int minDepth(TreeNode* root) {
        if(!root) return 0;
        if(!root->left && !root->right) return 1;
        int depth = INT32_MAX;
        if(root->left) depth = min(depth, minDepth(root->left));
        if(root->right) depth = min(depth, minDepth(root->right));
        return 1 + depth;
    }
};
```

完全二叉树的节点个数 #222

```
// 1. 递归遍历 O(n) O(logn)
class Solution {
public:
    int countNodes(TreeNode* root) {
        if(!root) return 0;
        return 1 + countNodes(root->left) + countNodes(root->right);
    }
};
// 2. 分别递归左孩子，和右孩子，递归到某一深度一定会有左孩子或者右孩子为满二叉树 O(logn *
logn)?? O(logn)
class Solution {
public:
    int countNodes(TreeNode* root) {
        if(!root) return 0;
        TreeNode* left = root->left;
        TreeNode* right = root->right;
        int leftDepth = 0, rightDepth = 0;
        while(left) {
            leftDepth++;
            left = left->left;
        }
        while(right) {
            rightDepth++;
            right = right->right;
        }
        if(leftDepth == rightDepth) {
            return pow(2, leftDepth + 1) - 1;
        }
        return 1 + countNodes(root->left) + countNodes(root->right);
    }
};
```

判断平衡二叉树 #110

```
class Solution {
public:
    int getDepth(TreeNode* root) {
        if(!root) return 0;
        int leftDepth = getDepth(root->left);
        int rightDepth = getDepth(root->right);
        if(leftDepth == -1 || rightDepth == -1) return -1;
        if(abs(leftDepth - rightDepth) <= 1) return 1 + max(leftDepth,
rightDepth);
        return -1;
    }
    bool isBalanced(TreeNode* root) {
        int depth = getDepth(root);
        return depth == -1 ? false : true;
    }
};
```

二叉树的所有路径 #257

```
class Solution {
public:
    void traverse(TreeNode* node, vector<int>& path, vector<string>& res) {
        path.push_back(node->val);
        if(!node->left && !node->right) {  // 1. 前序遍历
            string str;
            int i = 0;
            for(; i < path.size() - 1; i++) {
                str += to_string(path[i]);
                str += "->";
            }
            str += to_string(path[i]);
            res.push_back(str);
        }

        if(node->left) {
            traverse(node->left, path, res);
            path.pop_back();
        }
        if(node->right) {
            traverse(node->right, path, res);
            path.pop_back();
        }
    }
    vector<string> binaryTreePaths(TreeNode* root) {
        if(!root) return {};
        vector<int> path;
        vector<string> res;
        traverse(root, path, res);
        return res;
    }
};
```

左子叶之和 #404

```
// 1. 不能通过当前节点判断是否为左子叶，要通过父节点去判断
class Solution {
public:
    int sum;
    void traverse(TreeNode* root) {
        if(!root) return;
        if(root->left && !root->left->left && !root->left->right) {
            sum += root->left->val;
        }
        traverse(root->left);
        traverse(root->right);
    }
    int sumOfLeftLeaves(TreeNode* root) {
        if(!root) return 0;
        sum = 0;
        traverse(root);
        return sum;
    }
};
```

```
// 2. 后序遍历
class Solution {
public:
    int sumOfLeftLeaves(TreeNode* root) {
        if (root == NULL) return 0;

        int leftValue = sumOfLeftLeaves(root->left);    // 左
        int rightValue = sumOfLeftLeaves(root->right);  // 右
                                                        // 中
        int midValue = 0;
        if (root->left && !root->left->left && !root->left->right) { // 中
            midValue = root->left->val;
        }
        int sum = midValue + leftValue + rightValue;
        return sum;
    }
};
```

树左下角的值 #513

```
// 1. 递归遍历（回溯）
class Solution {
public:
    int maxDepth = -1;
    int maxVal;
    void traverse(TreeNode* root, int leftDepth) {
        if(!root->left && !root->right) {
            if(leftDepth > maxDepth) {  //保证首先遍历最后一行的最左边元素，而且只进入循环这一次
                maxDepth = leftDepth;
                maxVal = root->val;
            }
        }
        if(root->left) {
            leftDepth++;
            traverse(root->left, leftDepth);
            leftDepth--;
        }
        if(root->right) {
            leftDepth++;
            traverse(root->right, leftDepth);
            leftDepth--;
        }
    }
    int findBottomLeftValue(TreeNode* root) {
        if(!root) return -1;
        int leftDepth = 0;
        traverse(root, leftDepth);
        return maxVal;
    }
};
// 2. 层序遍历
class Solution {
public:
    int findBottomLeftValue(TreeNode* root) {
        if(!root) return -1;
        queue<TreeNode*> que;
        que.push(root);
        int val = 0;
        while(!que.empty()) {
            int size = que.size();
            for(int i = 0; i < size; i++) {
                TreeNode* node = que.front();
                que.pop();
                if(i == 0) val = node->val;  //只保存每一行的第一个元素，最后结束代表的就是最后一行最左边元素

                if(node->left) que.push(node->left);
                if(node->right) que.push(node->right);
            }
        }
        return val;

    }
};
```

路经总和 #112

```
class Solution {
public:
    bool traverse(TreeNode* root, int sum) {
        if(!root->left && !root->right) {
            if(sum == 0) return true;
        }
        if(root->left) {
            sum -= root->left->val;  // 回溯的时候，前后都要写好，因为不是全局变量，返回时在里边函数做的操作相当于没做
            if(traverse(root->left, sum)) return true;  // 一定要判断一下，否则后边的false会覆盖叶子节点返回的true
            sum += root->left->val;
        }
        if(root->right) {
            sum -= root->right->val;
            if(traverse(root->right, sum)) return true;
            sum += root->right->val;
        }
        return false;
    }
    bool hasPathSum(TreeNode* root, int targetSum) {
        if(!root) return false;
        return traverse(root, targetSum - root->val);   // 要提前减去root->val，traverse里不用先减了
    }
};
// 也可以定义全局变量和全局标志flag
```

路经总和II #113

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    void traverse(TreeNode* root, int sum) {
        if(!root->left && !root->right) {
            if(sum == 0) {
                res.push_back(path);
                return;
            }
        }
        if(root->left) {
            sum -= root->left->val;
            path.push_back(root->left->val);
            traverse(root->left, sum);
            sum += root->left->val;
            path.pop_back();
        }
        if(root->right) {
            sum -= root->right->val;
            path.push_back(root->right->val);
            traverse(root->right, sum);
            sum += root->right->val;
            path.pop_back();
        }
    }
    vector<vector<int>> pathSum(TreeNode* root, int targetSum) {
        res.clear();
        path.clear();
        if(!root) return res;

        path.push_back(root->val);
        traverse(root, targetSum - root->val);
        return res;
    }
};
```

中序和后序构造树 #106

```
class Solution {
public:
    TreeNode* traverse(vector<int>& inorder, int in_begin, int in_end,
vector<int>& postorder, int post_begin, int post_end) {
        // 注意不要使用 x.size(),下标不要用0或者x.size()-1 之类的
        if(post_begin == post_end) return nullptr;  // 一定要对空数组的判断

        int rootVal = postorder[post_end - 1];
        TreeNode* root = new TreeNode(rootVal);

        if(post_end - post_begin == 1) return root;

        int delimiterIndex;
        for(delimiterIndex = in_begin; delimiterIndex < in_end;
delimiterIndex++) {
```

```
            if(inorder[delimiterIndex] == rootVal) break;
        }

        root->left = traverse(inorder, in_begin, delimiterIndex, postorder,
post_begin, post_begin + delimiterIndex - in_begin);
        root->right = traverse(inorder, delimiterIndex + 1, in_end, postorder,
post_begin + delimiterIndex - in_begin, post_end - 1);
        return root;
    }
    TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
        if(inorder.size() == 0 || postorder.size() == 0) return nullptr;
        // 坚持左闭右开的区间
        return traverse(inorder, 0, inorder.size(), postorder, 0,
postorder.size());
    }
};
```

前序和中序构造树 #105

```
class Solution {
public:
    TreeNode* traverse(vector<int>& preorder, int pre_begin, int pre_end,
vector<int>& inorder, int in_begin, int in_end) {
        // 注意不要使用 x.size(),下标不要用0或者x.size()-1 之类的
        if(pre_begin == pre_end) return nullptr;  // 不要使用 preorder.size()

        int rootVal = preorder[pre_begin];  // 不要使用下标[0]
        TreeNode* root = new TreeNode(rootVal);

        if(pre_end - pre_begin == 1) return root;

        int delimiterIndex;
        for(delimiterIndex = in_begin; delimiterIndex < in_end;
delimiterIndex++) {
            if(inorder[delimiterIndex] == rootVal) break;
        }

        root->left = traverse(preorder, pre_begin + 1, pre_begin + 1 +
delimiterIndex - in_begin, inorder, in_begin, delimiterIndex);  // 起始位置不要用1,
而是pre_begin+1
        root->right = traverse(preorder, pre_begin + 1 + delimiterIndex -
in_begin, pre_end, inorder, delimiterIndex + 1, in_end);
        return root;
    }
    TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
        if(preorder.size() == 0 || inorder.size() == 0) return nullptr;
        return traverse(preorder, 0, preorder.size(), inorder, 0,
inorder.size());
    }
};
```

最大二叉树 #654

```
class Solution {
public:
    TreeNode* traverse(vector<int>& nums, int begin, int end) {
        int size = end - begin;
        if(size == 0) return nullptr;
        else if(size == 1) return new TreeNode(nums[begin]);

        int maxIndex = begin;
        for(int i = begin; i < end; i++) {
            if(nums[i] > nums[maxIndex]) maxIndex = i;
        }
        TreeNode* root = new TreeNode(nums[maxIndex]);
        root->left = traverse(nums, begin, maxIndex);
        root->right = traverse(nums, maxIndex + 1, end);

        return root;
    }
    TreeNode* constructMaximumBinaryTree(vector<int>& nums) {
        return traverse(nums, 0, nums.size());
    }
};
```

合并二叉树 #617

```
class Solution {
public:
    TreeNode* mergeTrees(TreeNode* root1, TreeNode* root2) {
        if(!root1 && !root2) return nullptr;
        else if(root1 && !root2) return root1;
        else if(!root1 && root2) return root2;

        root1->val = root1->val + root2->val;  // 重复利用root1
        root1->left = mergeTrees(root1->left, root2->left);  // 传入两个数的参数
        root1->right = mergeTrees(root1->right, root2->right);

        return root1;
    }
};
```

BST的搜索 #700

BST-二叉搜索树

```
// 1.递归
class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int val) {
        if(!root) return nullptr;
        if(root->val == val) return root;
        else if(root->val > val) return searchBST(root->left, val);
        return searchBST(root->right, val);
    }
};
```

```
// 2.迭代
class Solution {
public:
    TreeNode* searchBST(TreeNode* root, int val) {
        while(root) {
            if(root->val > val) root = root->left;
            else if(root->val < val) root = root->right;
            else return root;
        }
        return nullptr;
    }
};
```

验证BST #98

```

class Solution {
public:
    TreeNode* pre = nullptr;  // 记录前一个节点，方便比较，否则有INT_MIN值，不好比较
    bool isValidBST(TreeNode* root) {
        if(!root) return true;

        bool left = isValidBST(root->left);

        if((pre != nullptr) && (pre->val >= root->val)) return false;  // 一定是
>=,因为二叉搜索树不能有重复节点
        pre = root;

        bool right = isValidBST(root->right);

        return left && right;
    }
};
// 也可以定义 long long maxVal = LONG_MIN 方便比较，或者定义一个vector最后判断是否有序。
```

BST最小绝对差 #530

```
// 1. 保存前一个节点pre
class Solution {
public:
    TreeNode* pre = nullptr;
    int res = INT_MAX;
    void traverse(TreeNode* root) {
        if(!root) return;
        traverse(root->left);

        if((pre != nullptr) && (root->val - pre->val < res)) res = root->val -
pre->val;
        pre = root;

        traverse(root->right);
    }

    int getMinimumDifference(TreeNode* root) {
        traverse(root);
        return res;
    }
};

class Solution {
public:
    vector<int> vec;
    void traverse(TreeNode* root) {
        if(!root) return;
        traverse(root->left);
        vec.push_back(root->val);
        traverse(root->right);
    }
    int getMinimumDifference(TreeNode* root) {
        traverse(root);
        int min = INT_MAX;
        for(int i = 0; i < vec.size() - 1; i++) {
            int tmp = vec[i + 1] - vec[i];
            if(tmp < min) min = tmp;
        }
        return min;
    }
};
```

BST的众数 #501

```
class Solution {
public:
    vector<int> res;
    int max = 0;
    int cnt = 0;
    TreeNode* pre = nullptr;

    void traverse(TreeNode* root) {
        if(!root) return;
        traverse(root->left);

        if(pre == nullptr) {
            cnt = 1;
        } else if(pre->val == root->val) {
            cnt++;
        } else {
            cnt = 1;
        }
        pre = root;  // 一定记得更新

        if(cnt == max) {
            res.push_back(root->val);
        } else if(cnt > max) {
            res.clear();
            max = cnt;
            res.push_back(root->val);
        }

        traverse(root->right);
    }
    vector<int> findMode(TreeNode* root) {
        traverse(root);
        return res;
    }
};
```

二叉树的最近公共祖先 #236

```
// 后序遍历，回溯
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(root == p || root == q || root == nullptr) return root;

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

        // 四种情况
        if(left && right) return root;
        if(left && !right) return left;
        if(!left && right) return right;
        else return nullptr;
    }
};
```

二叉搜索树的最近公共祖先 #235

```
// 如果两个节点值都小于根节点，说明他们都在根节点的左子树上
// 如果两个节点值都大于根节点，说明他们都在根节点的右子树上
// 如果一个节点值大于根节点，一个节点值小于根节点，说明他们他们在根节点的异侧，那么根节点就是他们的最近公共祖先节点
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(!root) return nullptr;

        if((p->val < root->val) && (q->val < root->val)) return lowestCommonAncestor(root->left, p, q);
        else if((p->val > root->val) && (q->val > root->val)) return lowestCommonAncestor(root->right, p, q);
        else return root;
    }
};
```

BST插入 #701

```
// 1.
class Solution {
public:
    TreeNode* insertIntoBST(TreeNode* root, int val) {
        TreeNode* node = new TreeNode(val);
        if(!root) return node;

        TreeNode* cur = root;
        TreeNode* pre;
        while(cur) {
            pre = cur;
            if(val < cur->val) cur = cur->left;
            else cur = cur->right;
        }
        if(val < pre->val) pre->left = node;
        else pre->right = node;
        return root;
    }
};
// 2. 通过递归函数返回值完成新加入节点的父子关系赋值操作
class Solution {
public:
    TreeNode* insertIntoBST(TreeNode* root, int val) {
        if(!root) {
            TreeNode* node = new TreeNode(val);
            return node;
        }

        if(val < root->val) root->left = insertIntoBST(root->left, val);  // 串联起来了
        else root->right = insertIntoBST(root->right, val);

        return root;
    }
};
```

BST的删除 #450

```
// 由于BST的性质，最多只删除一个节点，因为没有重复的节点
class Solution {
public:
    TreeNode* deleteNode(TreeNode* root, int key) {
        if(!root) return nullptr;

        if(root->val == key) {
            if(!root->left && !root->right) {
                delete root;
                return nullptr;
            } else if(!root->left) {
                TreeNode* tmp = root->right;
                delete root;
                return tmp;
            } else if(!root->right) {
                TreeNode* tmp = root->left;
                delete root;
                return tmp;
            } else {
                 // 都不空，因为是删除一个节点，不能舍弃其他节点，所以比较麻烦，跟下一个题目进行对比
                TreeNode* cur = root->right;
                while(cur->left) {
                    cur = cur->left;
                }
                cur->left = root->left;
                TreeNode* tmp = root;
                root = root->right;
                delete tmp;
                return root;
            }
        } else if(root->val > key) {
            root->left = deleteNode(root->left, key);
        } else {
            root->right = deleteNode(root->right, key);
        }
        return root;
    }
};
```

修剪BST #669

```
class Solution {
public:
    TreeNode* trimBST(TreeNode* root, int low, int high) {
        if(!root) return nullptr;

        // 跟删除特定的值不同，删除不在某个区间的值，根据值大小的判断可以舍弃某一半的子树，返回符合条件的另一半子树
        if(root->val < low) {
            TreeNode* right = trimBST(root->right, low, high);
            return right;
        } else if(root->val > high) {
            TreeNode*  left = trimBST(root->left, low, high);
            return left;
        }

        root->left = trimBST(root->left, low, high);  // root->left接入符合条件的左孩子
        root->right = trimBST(root->right, low, high);  // root->right接入符合条件的右孩子
        return root;
    }
};
```

有序数组转为平衡BST #108

```
// 每次以数组中间位置的元素为分割点，自然就是平衡二叉搜索树
class Solution {
public:
    TreeNode* traverse(vector<int>& nums, int begin, int end) {
        int size = end - begin;
        if(size == 0) return nullptr;
        if(size == 1) return new TreeNode(nums[begin]);

        int mid = begin + size / 2;
        TreeNode* root = new TreeNode(nums[mid]);
        root->left = traverse(nums, begin, mid);
        root->right = traverse(nums, mid + 1, end);

        return root;
    }
    TreeNode* sortedArrayToBST(vector<int>& nums) {
        return traverse(nums, 0, nums.size());
    }
};
```

BST转为累加树 #538

```
// 逻辑是这样的
// 首先明确遍历的顺序是【逆中序】遍历
// 其次，BST的中序遍历是有序的，我们要做的中间结点的逻辑就是加上前一个结点的值，定义一个pre保存前一个结点的值
// 然后因为是逆中序遍历，所以相当于是倒着往前加
class Solution {
public:
    int pre = 0;
    void traverse(TreeNode* root) {
        if(!root) return;

        traverse(root->right);
        root->val += pre;
        pre = root->val;
        traverse(root->left);
    }
    TreeNode* convertBST(TreeNode* root) {
        traverse(root);
        return root;
    }
};
```

☆回溯

递归要素

- 确定参数和返回值
- 确定终止条件
- 确定单层递归逻辑

二叉树的最大路径和 #124

```
class Solution {
private:
    int maxSum = INT_MIN;
public:
    int dfs(TreeNode *node){
        if(node == nullptr) return 0;

        int leftMax = max(dfs(node->left), 0);
        int rightMax = max(dfs(node->right), 0);

        int newPath = node->val + leftMax + rightMax;
        maxSum = max(maxSum, newPath);
        return node->val + max(leftMax, rightMax);
    }
    int maxPathSum(TreeNode* root) {
        dfs(root);
        return maxSum;
    }
};
```

组合 #77

```
//  回溯精神：for循环横向遍历，递归纵向遍历，回溯不断调整结果集
class Solution {

public:
    vector<vector<int>> res;
    vector<int> path;
    void backtracking(int n, int k, int begin) {
        if(path.size() == k) {
            res.push_back(path);
            return;
        }

        // for(int i = begin; i <= n; i++) {
        for(int i = begin; i <= n - (k - path.size()) + 1; i++) { // 优化，剪枝
            path.push_back(i);
            backtracking(n, k, i + 1);
            path.pop_back();
        }
        return;
    }
    vector<vector<int>> combine(int n, int k) {
        res.clear();
        path.clear();
        backtracking(n, k, 1);
        return res;
    }
};
```

组合总和III #216

找出所有相加之和为 n 的 k 个数的组合。组合中只允许含有 1 - 9 的正整数.

每种组合中不存在重复的数字。

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    int curSum = 0;
    void backtracking(int k, int n, int begin) {
        if(curSum > n) {
            // 这里千万不要回溯，直接return就好，因为for循环会统一回溯，否则会多pop元素了
            // curSum -= path[path.size() - 1];
            // path.pop_back();
            return;
        }
        if(path.size() == k) {
            if(curSum == n) {
                res.push_back(path);
            }
            return;
        }
        // 这样写也对，但是path.size()会超过k，会进行下边无意义的多余的循环
        // if(path.size() == k && curSum == n) {
```

```
        //     res.push_back(path);
        //     return;
        // }

        for(int i = begin; i <= 9 - (k - path.size()) + 1; i++) {
            curSum += i;
            path.push_back(i);
            backtracking(k, n, i + 1);
            curSum -= i;
            path.pop_back();
        }
        return;
    }
    vector<vector<int>> combinationSum3(int k, int n) {
        backtracking(k, n, 1);
        return res;
    }
};
```

电话号码的字母组合 #17

```
// 这个是多个集合求组合，上一个题是一个集合求组合。
class Solution {
public:
    // 字母表
    const string letterMap[10] = {
        "",
        "",
        "abc",
        "def",
        "ghi",
        "jkl",
        "mno",
        "pqrs",
        "tuv",
        "wxyz"
    };
    vector<string> res;
    string str;
    // 没有begin，而是index代表遍历的是第几个集合，每个集合从0开始遍历
    void backtracking(const string& digits, int index) {
        if(index == digits.size()) {
            res.push_back(str);
            return;
        }

        int letter = digits[index] - '0';
        string letters = letterMap[letter];
        for(int i = 0; i < letters.size(); i++) {
            str.push_back(letters[i]);
            backtracking(digits, index + 1);
            str.pop_back();
        }
        return;
    }
    vector<string> letterCombinations(string digits) {
        if(digits.size() == 0) return res;
        backtracking(digits, 0);
        return res;
    }
};
```

组合总和 #39

无重复元素数组 candidates, 目标数 target, 找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的数字可以重复取。

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    int curSum = 0;
    void backtracking(vector<int>& candidates, int target, int index) {
        if(curSum > target) return;
        if(curSum == target) {
            res.push_back(path);
            return;
        }

        for(int i = index; i < candidates.size(); i++) {
            curSum += candidates[i];
            path.push_back(candidates[i]);
            backtracking(candidates, target, i); // 不是 index+1, 也不是 i+1, 而是 i, 代表可以重复选取
            curSum -= candidates[i];
            path.pop_back();
        }
    }
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        backtracking(candidates, target, 0);
        return res;
    }
};
```

组合总和II #40

有重复数组 candidates, 目标数 target, 找出 candidates 中所有可以使数字和为 target 的组合。

candidates 中的每个数字在每个组合中只能使用一次, 解集不能包含重复的组合。

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    int curSum = 0;
    void backtracking(vector<int>& candidates, int target, int index) {
        if(curSum > target) return;
        if(curSum == target) {
            res.push_back(path);
            return;
        }
        for(int i = index; i < candidates.size(); i++) {
            if(i > index && candidates[i] == candidates[i - 1]) continue;  // 同一个树层，不可以重复取
            curSum += candidates[i];
            path.push_back(candidates[i]);
            backtracking(candidates, target, i + 1);
            curSum -= candidates[i];
            path.pop_back();
        }
    }
    vector<vector<int>> combinationSum2(vector<int>& candidates, int target) {
        // 必须要先进行排序，后边去重才是有效的
        sort(candidates.begin(), candidates.end());
        backtracking(candidates, target, 0);
        return res;
    }
};
```

分割回文串 #131

```
class Solution {
public:
    vector<vector<string>> res;
    vector<string> path;
    bool isPalindrome(const string& s, int start, int end) {  // 要定义start, end
        for(int i = start, j = end; i < j; i++, j--) {
            if(s[i] != s[j]) return false;
        }
        return true;
    }
    // index是切割的位置，表示在s[index]这个元素后边进行切割，当index=s.size()则是切割到末尾了
    void backtracking(const string& s, int index) {  // 定义为 const string& 是引用，节省空间和时间
        if(index >= s.size()) {
            res.push_back(path);
            return;
        }

        for(int i = index; i < s.size(); i++) {
            if(isPalindrome(s, index, i)) {  // [index, i] 闭区间的子串
                path.push_back(s.substr(index, i - index + 1));  // string.substr(index,len) 第二个参数是长度
            } else {
                continue;
            }

            backtracking(s, i + 1);
            path.pop_back();
        }
    }
    vector<vector<string>> partition(string s) {
        backtracking(s, 0);
        return res;
    }
};
```

复原IP地址 #93

```
class Solution {
public:
    vector<string> res;
    vector<string> path;
    void backtracking(const string& s, int index) {
        if(path.size() == 4 && index >= s.size()) {  // 一定别忘了必须有且只有四段，而且两个条件必须同时满足
            string str;
            for(string ss : path) {
                str += ss + '.';
            }
            str.pop_back();
            res.push_back(str);
            return;
        }

        for(int i = index; i < s.size(); i++) {
            if(isValid(s, index, i)) {
                path.push_back(s.substr(index, i - index + 1));
            } else {
                break;  // 如果不合法，直接跳出循环即可，没必要continue了
            }

            backtracking(s, i + 1);
            path.pop_back();
        }
    }
    bool isValid(const string& s, int begin, int end) {
        if(begin > end) return false;  // 别忘了判断是否为空
        if(s[begin] == '0' && begin != end) return false;

        int num = 0;
        for(int i = begin; i <= end; i++) {
            if(s[i] > '9' || s[i] < '0') return false;
            num = (s[i] - '0') + num * 10;
            if(num > 255) return false;
        }
        return true;
    }

    vector<string> restoreIpAddresses(string s) {
        if(s.size() > 12) return res;  // 剪枝操作
        backtracking(s, 0);
        return res;
    }
};
```

子集 #78

无重复元素数组nums，返回数组所有子集

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    void backtracking(vector<int>& nums, int index) {
        res.push_back(path);  // 提前加
        if(index >= nums.size()) {
            return;
        }

        for(int i = index; 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 res;
    }
};
```

子集II #90

有重复元素的整数数组 nums，返回该数组所有可能的子集（幂集）。

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    void backtracking(vector<int>& nums, int index) {
        res.push_back(path);
        if(index >= nums.size()) {
            return;
        }

        for(ini = index; i < nums.size(); i++) {
            if(i > index && nums[i] == nums[i - 1]) continue;  // 去重
            path.push_back(nums[i]);
            backtracking(nums, i + 1);
            path.pop_back();
        }
    }
    vector<vector<int>> subsetsWithDup(vector<int>& nums) {
        if(nums.size() == 0) return res;
        sort(nums.begin(), nums.end());
        backtracking(nums, 0);
        return res;
    }
};
```

递增子序列 #491

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    void backtracking(vector<int>& nums, int index) {
        if(path.size() >= 2) res.push_back(path);
        if(index >= nums.size()) return;

        unordered_set<int> uset;  // 记录本层（树层）元素是否重用过
        for(int i = index; i < nums.size(); i++) {
            // 一定要判断 path.empty()，不要用 i>index 去判断
            // i>index 的时候：
            // (1)树退回到第一层的时候，i>index，但是path是空的，path.back()会报错或者返回一个很大的数
            // (2)树往下递归的时候，选取集合第一个元素，i==index 不会进入if()，但是如果元素小于path.back()会加到path里
            if((!path.empty() && nums[i] < path.back()) || (uset.find(nums[i]) != uset.end())) continue;
            uset.insert(nums[i]);
            path.push_back(nums[i]);
            backtracking(nums, i + 1);
            path.pop_back();
        }
    }
    vector<vector<int>> findSubsequences(vector<int>& nums) {
        backtracking(nums, 0);
        return res;
    }
};
```

全排列 #46

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    void backtracking(vector<int>& nums, vector<bool>& used) {
        if (path.size() == nums.size()) {
            res.push_back(path);
            return;
        }

        for (int i = 0; i < nums.size(); i++) {
            if (used[i] == true) continue;
            used[i] = true;  // 不要写成 ==
            path.push_back(nums[i]);
            backtracking(nums, used);
            used[i] = false;
            path.pop_back();
        }
    }
    vector<vector<int>> permute(vector<int>& nums) {
        vector<bool> used(nums.size(), false);
        backtracking(nums, used);
        return res;
    }
};
```

全排列II #47

```
class Solution {
public:
    vector<vector<int>> res;
    vector<int> path;
    void backtracking(vector<int>& nums, vector<bool>& used) {
        if (path.size() == nums.size()) {
            res.push_back(path);
            return;
        }

        // used[i - 1] == true 代表当前这个树枝用过i-1这个元素
        // used[i - 1] == false 代表当前这个树层用过i-1这个元素
        // 因为同一个树层，在"弹"回来的时候，前一个元素的used置为了false，而且只能判断前一个元素是否用过
        for (int i = 0; i < nums.size(); i++) {
            if(used[i] == true) continue;
            if (i > 0 && nums[i] == nums[i - 1] && used[i - 1] == false) continue;
            used[i] = true;
            path.push_back(nums[i]);
            backtracking(nums, used);
            path.pop_back();
            used[i] = false;
        }
    }
    vector<vector<int>> permuteUnique(vector<int>& nums) {
        vector<bool> used(nums.size(), false);
        sort(nums.begin(), nums.end());  // 结果集需要去重的都需要先排序
        backtracking(nums, used);
        return res;
    }
};
```

重新安排行程 #332

```
class Solution {
public:
    unordered_map<string, map<string, int>> targets;
    vector<string> res;
    bool backtracking(int ticketsNum) {
        if (res.size() == ticketsNum + 1) return true;  // 欧拉回路，res.size() = 顶点数 = 边数 + 1 = ticketsNum + 1

        // &引用，一定是遍历res中最末尾的节点对应的targets
        // const string 是因为 map 中的 key 是不能改变的
        for (pair<const string, int>& tar : targets[res.back()]) {
            if (tar.second > 0) {
                tar.second--;
                res.push_back(tar.first);
                if(backtracking(ticketsNum)) return true;  // 这个很关键，只需要找到一条合适的到达叶子节点的路径
                res.pop_back();
                tar.second++;  // 别忘了
            }
        }
        return false;
    }
    vector<string> findItinerary(vector<vector<string>>& tickets) {
        for(const vector<string>& ticket : tickets) {  // const
            targets[ticket[0]][ticket[1]]++;
        }
        res.push_back("JFK");
        backtracking(tickets.size());
        return res;
    }
};
```

N皇后 #51

```
class Solution {
public:
    vector<vector<string>> res;
    vector<string> chessboard;
    void backtracking(int n, int row) {
        if (row == n) {
            res.push_back(chessboard);
            return;
        }

        // row代表行，深度遍历；col代表列，for层遍历
        for (int col = 0; col < n; col++) {
            if (isValid(n, row, col)) {
                chessboard[row][col] = 'Q';
                backtracking(n, row + 1);
                chessboard[row][col] = '.';
            }
        }

    }
    bool isValid(int n, int row, int col) {
        // 上方的列的方向，下方不用考虑，因为会回溯回来
        for (int i = 0; i < row; i++) {
            if (chessboard[i][col] == 'Q') return false;
        }
       // 左斜上45°方向，右斜下45°方向不用考虑
        for (int i = row - 1, j = col - 1; i >= 0 && j >= 0; i--, j--) {
            if(chessboard[i][j] == 'Q') return false;
        }
        // 右斜上45°方向，左斜下45°方向不用考虑
        for (int i = row - 1, j = col + 1; i >= 0 && j < n; i--, j++) {
            if(chessboard[i][j] == 'Q') return false;
        }
        return true;

    }
    vector<vector<string>> solveNQueens(int n) {
        chessboard = vector<string>(n, string(n, '.'));
        backtracking(n, 0);
        return res;
    }
};
```

解数独 #37

```
class Solution {
public:
    // 符合条件的才返回，其他的丢弃，所以有bool返回值
    bool backtracking(vector<vector<char>>& board) {
        // N皇后:只需要一层for循环遍历一行，递归来来遍历列，因为每一行每一列只放一个皇后
        // 数独:二维递归，先遍历二维的结构，两个for循环嵌套一个递归，递归遍历数字1-9
        //      不需要返回值，因为每次递归都比之前多填了一个数，当数填满之后就自动停止了
        for(int row = 0; row < 9; row++) {
            for(int col = 0; col < 9; col++) {
                if(board[row][col] != '.') continue;
                for(char ch = '1'; ch <= '9'; ch++) {
                    if(isValid(row, col, ch, board)) {
                        board[row][col] = ch;
                        if(backtracking(board)) return true;
                        board[row][col] = '.';
                    }
                }
                return false;  // 尝试了1-9个数都不行，那么说明无解
            }
        }
        return true;
    }
    bool isValid(int row, int col,char ch, vector<vector<char>>& board) {
        // 行列
        for(int i = 0; i < 9; i++) {
            if(board[row][i] == ch) return false;
            if(board[i][col] == ch) return false;
        }

        // 3x3格子
        int startRow = row / 3 * 3;
        int startCol = col / 3 * 3;
        for(int i = startRow; i < startRow + 3; i++) {
            for(int j = startCol; j < startCol + 3; j++) {
                if(board[i][j] == ch) return false;
            }
        }
        return true;
    }
    void solveSudoku(vector<vector<char>>& board) {
        backtracking(board);
        return;
    }
};
```

☆贪心

分发饼干 #455

```
class Solution {
public:
    int findContentChildren(vector<int>& g, vector<int>& s) {
        sort(g.begin(), g.end());
        sort(s.begin(), s.end());

        int index_g = 0;
        // 遍历孩子，小饼干先喂饱胃口小的孩子
        // index_g 也是一个技巧，不要写两个循环
        for(int i = 0; i < s.size(); i++) {
            if(index_g < g.size() && g[index_g] <= s[i]) {
                index_g++;
            }
        }
        return index_g;
    }
};
```

摆动序列 #376

```
class Solution {
public:
    int wiggleMaxLength(vector<int>& nums) {
        int curDiff = 0;
        int preDiff = 0;  // 为了好比较
        int res = 1;  // 默认为1，因为长度为2的数组摆动序列长度为2

        for(int i = 0; i < nums.size() -  1; i++) {
            curDiff = nums[i + 1] - nums[i];
            // 这里没有 curDiff=0 的情况，说明自动忽略了平坡的情况
            if((curDiff > 0 && preDiff <= 0) || (curDiff < 0 && preDiff >= 0)) {
                res++;
                preDiff = curDiff;
            }
        }

        return res;
    }
};
```

最大子数组和 #53

```
class Solution {
public:
    int maxSubArray(vector<int>& nums) {
        if(nums.size() == 1) return nums[0];
        int sum = 0;
        int res = INT_MIN;
        for(int i = 0; i < nums.size(); i++) {
            sum += nums[i];
            if(sum > res) {
                res = sum;
            }
            if(sum < 0) sum = 0;  // 如果当前和小于0，一定重新开始计，因为加负数一定更小
        }

        return res;
    }
};
```

买卖股票的最佳时机II #122

```
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if(prices.size() == 1) return 0;
        int res = 0;
        for(int i = 1; i < prices.size(); i++) {
            res += max(0, prices[i] - prices[i - 1]);  // 只收集正利润
        }
        return res;
    }
};
```

跳跃游戏 #55

```
class Solution {
public:
    bool canJump(vector<int>& nums) {
        int index = 0;
        for(int i = 0; i < nums.size(); i++) {
            if(index < i) return false;
            index = max(index, i + nums[i]);  // 每次移动，更新跳跃的最大范围

        }
        return true;
    }
};
```

跳跃游戏II #45

```
class Solution {
public:
    int jump(vector<int>& nums) {
        if (nums.size() == 1) return 0;
        int curDistance = 0, nextDistance = 0;
        int step = 0;

        // 关键在于不知道当前要跳多少步，这个方法比较巧妙，不去考虑跳多少步，而是决定在什么时候
加步数
        // 还是一个一个遍历，遍历的时候不断更新覆盖范围，当前坐标跟当前覆盖范围相等的时候，就要
加步数了
        for (int i = 0; i < nums.size(); i++) {
            nextDistance = max(nums[i] + i, nextDistance);
            if (i == curDistance) {
                step++;
                curDistance = nextDistance;
            } else {
                continue;
            }
            if (nextDistance >= nums.size() - 1) break;
        }

        return step;
    }
};
```

K次取反后最大化的数组和 #1005

```
class Solution {
static bool cmp(int a, int b) {
    return abs(a) > abs(b);
}
public:
    int largestSumAfterKNegations(vector<int>& nums, int k) {
        int res = 0;
        sort(nums.begin(), nums.end(), cmp);  // 按照绝对值大小，从大到小排序
        for (int i = 0; i < nums.size(); i++) {
            if(nums[i] < 0 && k > 0) {  // 1. 局部最优，优先把绝对值大的负数转为正数
                nums[i] *= -1;
                k--;
            }
        }
        // while (k) {
        //     nums[nums.size() - 1] *= -1;
        //     k--;
        // }
        if(k % 2 == 1) nums[nums.size() - 1] *= -1;  // 2. 如果k仍然>0，说明负数都处理过了，那么在绝对值最小的数上进行重复翻转
        for(int a : nums) res += a;

        return res;
    }
};
```

加油站 #134

```
class Solution {
public:
    int canCompleteCircuit(vector<int>& gas, vector<int>& cost) {
        int sum = 0;
        int min_gas = INT_MAX;
        int min_index;

        // 想象折线图，找到折线图的最低点，将整个折线图移到x轴以上，x轴的交点即是最低点，最低点下一个点就是起始的点
        // 最低点的这个点的gas-cost肯定是你负的，否则不会是最低，所以从当前这个点开始肯定不行，要从下一个点开始
        // 无论最低点前边有多少个负数，从下一个节点开始会一点点弥补回来
        for (int i = 0; i < gas.size(); i++) {
            sum += gas[i] - cost[i];
            if (sum < min_gas) {
                min_gas = sum;
                min_index = i;
            }
        }

        return sum < 0 ? -1 : (min_index + 1) % gas.size();
    }
};
```

分发糖果 #135

```
class Solution {
public:
    int candy(vector<int>& ratings) {
        vector<int> candy(ratings.size(), 1);
        // 先从左往右遍历，如果右边比左边分数高，那么右边糖果数+1
        for (int i = 0; i < ratings.size() - 1; i++) {
            if (ratings[i + 1] > ratings[i]) {
                candy[i + 1] = candy[i] + 1;  // candy[i + 1]++ 这样写不对
            }
        }
        // 再从右往左遍历，而且要从后往前，如果左边比右边分数高
        // 那么左边糖果数等于max(之前更新的，右边+1),要考虑之前更新的值，就是为了保持之前的比较关系
        for (int i = ratings.size() - 1; i > 0; i--) {
            if (ratings[i - 1] > ratings[i]) {
                candy[i - 1] = max(candy[i - 1], candy[i] + 1);
            }
        }

        int res = 0;
        for (int a : candy) res += a;
        return res;
    }
};
```

柠檬水找零 #860

```
class Solution {
public:
    bool lemonadeChange(vector<int>& bills) {
        // 划分为三个情况去分析
        int five = 0, ten = 0, twenty = 0;
        for (int bill : bills) {
            if (bill == 5) {
                five++;
            } else if (bill == 10) {
                if (five <= 0) return false;
                five--;
                ten++;
            } else if (bill == 20) {
                // 付款20的时候，优先找5+10，其次5*3，因为5的作用更多，还可以应对付款10的情况
                if (five > 0 && ten > 0) {
                    five--;
                    ten--;
                    twenty++;
                } else if (five >= 3) {
                    five -= 3;
                    twenty++;
                } else return false;
            }
        }
        return true;
    }
};
```

根据身高重建队列 #406

```
class Solution {
public:
    static bool cmp(vector<int>& a, vector<int>& b) {
        if(a[0] == b[0]) return a[1] < b[1];  // 意思是a[1]如果小于b[1]的话，a[1]在前边
        return a[0] > b[0];  // [0]大的在前边
    }
    vector<vector<int>> reconstructQueue(vector<vector<int>>& people) {
        sort(people.begin(), people.end(), cmp);
        list<vector<int>> que;  // 底层实现是链表，插入速度更快

        for (int i = 0; i < people.size(); i++) {
            int k = people[i][1];
            list<vector<int>>::iterator it = que.begin();  // 注意语法
            while(k--) {
                it++;
            }
            que.insert(it, people[i]);
        }

        return vector<vector<int>>(que.begin(), que.end());
    }
};
```

用最少数量的箭引爆气球 #452

```
class Solution {
public:
    static bool cmp(vector<int>& a, vector<int>& b) {
        return a[0] < b[0];
    }
    int findMinArrowShots(vector<vector<int>>& points) {
        if (points.size() == 1) return 1;
        sort(points.begin(), points.end(), cmp);

        int res = 1;
        for (int i = 1; i < points.size(); i++) {
            if (points[i][0] > points[i - 1][1]) {  // 通过右边界判断，不挨着，需要箭数+1，注意不包含等于
                res++;
            } else {  // 如果挨着，右边界统一到右边气球的左坐标上，这样其实points[i]的区间就代表了重叠的区域
                points[i][1] = min(points[i - 1][1], points[i][1]);
            }
        }

        return res;
    }
};
```

无重叠区间 #435

```
class Solution {
public:
    static bool cmp(vector<int>& a, vector<int>& b) {
        return a[1] < b[1];  // 按照右边界，从小到大排序
    }
    int eraseOverlapIntervals(vector<vector<int>>& intervals) {
        if (intervals.size() == 1) return 0;

        sort(intervals.begin(), intervals.end(), cmp);
        int res = 1;
        int end = intervals[0][1];
        for (int i = 1; i < intervals.size(); i++) {  //  从左到右遍历
            if (intervals[i][0] >= end) {  // 计算非重叠区域的个数，重叠的自动跳过
                res++;
                end = intervals[i][1];
            }
        }
        return intervals.size() - res;
    }
};
```

划分字母区间 #763

```
class Solution {
public:
    vector<int> partitionLabels(string s) {
        int hash[26] = {0};
        for (int i = 0; i < s.size(); i++) {
            hash[s[i] - 'a'] = i;  // 记录当前字符出现的最远位置
        }

        int left = 0, right = 0;
        vector<int> res;
        for (int i = 0; i < s.size(); i++) {
            right = max(right, hash[s[i] - 'a']);  // 记录一段字符中最远的边界
            if (i == right) {  // 找到之前字符出现的最大出现位置和当前位置相等，即为一个片段
                res.push_back(right - left + 1);
                left = i + 1;
            }
        }

        return res;
    }
};
```

合并区间 #56

```
class Solution {
public:
    static bool cmp(vector<int>& a, vector<int>& b) {
        return a[0] < b[0];
    }
    vector<vector<int>> merge(vector<vector<int>>& intervals) {
        if (intervals.size() == 1) return intervals;
        sort(intervals.begin(), intervals.end(), cmp);

        vector<vector<int>> res;
        for (int i = 1; i < intervals.size(); i++) {
            if (intervals[i][0] <= intervals[i - 1][1]) {
                intervals[i][0] = min(intervals[i - 1][0], intervals[i][0]);
                intervals[i][1] = max(intervals[i - 1][1], intervals[i][1]);
            } else {
                res.push_back(intervals[i - 1]);
            }
        }
        res.push_back(intervals.back());

        return res;
    }
};
```

单调递增的数字 #738

```
class Solution {
public:
    int monotoneIncreasingDigits(int n) {
        if(n < 10) return n;

        string s = to_string(n);  // to_string()
        int flag = s.size();  // 记录需要赋值为9的最初位置，后边都要赋值为9
        for (int i = s.size() - 1; i > 0; i--) {
            if (s[i - 1] > s[i]) {
                flag = i;
                s[i - 1]--;
            }
        }

        for (int i = flag; i < s.size(); i++) {
            s[i] = '9';
        }

        return stoi(s);  // stoi()
    }
};
```

监控二叉树 #968

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode() : val(0), left(nullptr), right(nullptr) {}
 *     TreeNode(int x) : val(x), left(nullptr), right(nullptr) {}
 *     TreeNode(int x, TreeNode *left, TreeNode *right) : val(x), left(left), right(right) {}
 * };
 */
class Solution {
public:
    int res = 0;
    int traverse(TreeNode* root) {
        // 0：无覆盖
        // 1：有覆盖
        // 2：有摄像头
        if (root == nullptr) return 1;
        int left = traverse(root->left);
        int right = traverse(root->right);
        if (left == 1 && right == 1) {  //  11 两个孩子都覆盖，说明父节点肯定无覆盖，这里先不要急着加摄像头
            return 0;
        } else if (left == 0  || right == 0) {  // 00,01,02,10,20 只要有一个无覆盖，父节点一定要加摄像头，否则跨过去就落下了
            res++;
            return 2;
        } else {  // 12,21,22
            return 1;
        }
    }
    int minCameraCover(TreeNode* root) {
        if (traverse(root) == 0) {  // 处理根节点无覆盖的情况
            res++;
        }
        return res;
    }
};
```

☆动态规划

模板

- 确定dp数组和下标的含义
- 确定递推公式
- dp数组初始化
- 确定遍历顺序
- 举例推导dp数组

```
void bag_01() {
    vector<int> weight = {1, 3, 4};
    vector<int> value = {15, 20, 30};
    int bagweight = 4;

    // 二维数组-模板
    vector<vector<int>> dp(weight.size(), vector<int>(value.size(), 0));
    // 初始化
    for (int j = weight[0]; j <= bagweight; j++) {
        dp[0][j] = value[0];
    }
    // i,j 都从 1 开始遍历，因为下标为 0 的都初始化过了
    for (int i = 1; i < weight.size(); i++) {
        for (int j = 1; j <= bagweight; j++) {
            if (j < weight[i]) {  // 不选i
                dp[i][j] = dp[i - 1][j];
            } else {              // 选i
                dp[i][j] = max(dp[i - 1][j], dp[i - 1][j - weight[i]] +
value[i]);
            }
        }
    }

    // 一维数组-模板
    vector<int> dp(bagweight + 1, 0);
    for (int i = 0; i < weight.size(); i++) {
         // 背包重量的【遍历必须在里边】。如果放在外边，那么背包只放了一个物品；放在里边，相当
于考虑了 0...i 这些物品
         // 一维dp数组，用【倒序】，保证了物品不会重复放，而且本质还是对二维数组的遍历，右下角
的值依赖上一层左上角的值
         // j >= nums[i] 防止下标溢出
         for (int j = bagweight; j >= weight[i]; j--) {
             dp[j] = max(dp[j], dp[j - weight[i]] + value[i]);
         }
    }

    cout << dp[weight.size() - 1][bagweight] << endl;
}

int main() {
    bag_01();
}
```

斐波那契数 #509

```
class Solution {
public:
    int fib(int n) {
        if (n < 2) return n;

        int dp[n + 1];
        dp[0] = 0;
        dp[1] = 1;
        for (int i = 2; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }

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

爬楼梯 #70

```
class Solution {
public:
    int fib(int n) {
        if (n < 2) return n;

        int dp[n + 1];
        dp[0] = 0;
        dp[1] = 1;
        for (int i = 2; i <= n; i++) {
            dp[i] = dp[i - 1] + dp[i - 2];
        }

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

不同路径 #62

```
class Solution {
public:
    int uniquePaths(int m, int n) {
        if (m == 1 || n == 1) return 1;

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

不同路径II #63

```
class Solution {
public:
    int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
        int m = obstacleGrid.size();
        int n = obstacleGrid[0].size();

        if (obstacleGrid[0][0] == 1 || obstacleGrid[m - 1][n - 1]) return 0;
        vector<vector<int>> dp(m, vector<int>(n, 0));
        for (int i = 0; i < m && obstacleGrid[i][0] == 0; i++) dp[i][0] = 1;
        for (int j = 0; j < n && obstacleGrid[0][j] == 0; j++) dp[0][j] = 1;
        for (int i = 1; i < m; i++) {
            for (int j = 1; j < n; j++) {
                if (obstacleGrid[i][j] == 1) continue;
                dp[i][j] = dp[i - 1][j] + dp[i][j - 1];
            }
        }

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

整数拆分 #343

```
class Solution {
public:
    int integerBreak(int n) {
        vector<int> dp(n + 1, 0);
        dp[2] = 1;  // dp[0],dp[1]没有意义

        // i从3开始遍历，j代表从1-(i-1)进行拆分
        for (int i = 3; i <= n; i++) {
            for (int j = 1; j <= i - 1; j++) {
                // 1. 要比较dp[i]，因为这个循环可能随时变化
                // 2. 为什么要比较 (i-j)*j，因为 dp[i-j]里不包括拆分为 i-j 的情况
                // 3. 可以这样理解，(i-j)*j 是拆分为两个数的情况，dp[i-j]*j 是拆分为两个
数及以上的情况
                dp[i] = max(dp[i], max(dp[i - j] * j, (i - j) * j));
            }
        }

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

不同的二叉搜索树 #96

```
class Solution {
public:
    int numTrees(int n) {
        vector<int> dp(n + 1);
        dp[0] = 1;
        dp[1] = 1;

        // 1...i 一共 i 个节点
        for (int i = 2; i <= n; i++) {
            // 以 j 为头节点， 根据二叉搜索树规则，左子树为 j-1 个节点， 右子树为 i-j 个节
点
            for (int j = 1; j <= i; j++) {
                dp[i] += dp[j - 1] * dp[i - j];
            }
        }

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

分割等和子集 #416

```
class Solution {
public:
    bool canPartition(vector<int>& nums) {

        // 每个元素最大是100，数组长度最大是200，所以一般和最大为10000
        int sum = 0;
        for (int i = 0; i < nums.size(); i++) {
            sum += nums[i];
        }
        if (sum % 2 == 1) return false;
        int target = sum / 2;
        vector<int> dp(target, 0);

        // dp[j]代表背包重量为 j 时，可以得到的最大的value值，这里即最大的子集的和
        // 背包重量 j 最大为 target， dp[j]最大值时是 j，j取最大时也就是 target
        for (int i = 0; i < nums.size(); i++) {
            // 背包重量的【遍历必须在里边】。如果放在外边，那么背包只放了一个物品；放在里边，相当于考虑了 0...i 这些物品
            // 一维dp数组，用【倒序】，保证了物品不会重复放，而且本质还是对二维数组的遍历，右下角的值依赖上一层左上角的值
            // j >= nums[i] 防止下标溢出
            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;
        return false;
    }
};
```

最后一块石头的重量II #1049

```
class Solution {
public:
    int lastStoneWeightII(vector<int>& stones) {
        int sum = 0;
        for (int i = 0; i < stones.size(); i++) {
            sum += stones[i];
        }
        int target = sum / 2;
        vector<int> dp(target + 1, 0);

        for (int i = 0; i < stones.size(); i++) {
            for (int j = target; j >= stones[i]; j--) {
                dp[j] = max(dp[j], dp[j - stones[i]] + stones[i]);
            }
        }

        return sum - dp[target] * 2;
    }
};
```

目标和 #494

```
class Solution {
public:
    int findTargetSumWays(vector<int>& nums, int target) {
        int sum = 0;
        for (int num : nums) sum += num;
        if (abs(target) > sum) return 0;           // S有可能是负数
        if ((target + sum) % 2 == 1) return 0;
        int bagSize = (target + sum) / 2;

        // 初始化
        // dp[0]=1，不能为0，因为后边的累加都要依赖于dp[0]，可以解释为装满容量为0的背包有1种方法，也就是装0件物品
        // 其他的要初始化为0，否则会对累加的结果有影响
        vector<int> dp(bagSize + 1, 0);
        dp[0] = 1;
        for (int i = 0; i < nums.size(); i++) {
            // 求装满背包有几种方法的情况下，递推公式一般为 dp[j] = dp[j - nums[i]]
            for (int j = bagSize; j >= nums[i]; j--) {
                dp[j] += dp[j - nums[i]];
            }
        }

        return dp[bagSize];
    }
};
```

☆其他

螺旋矩阵 #59

☆单调栈

☆Error

```
1. vector.size() 返回的是无符号数，vector.size()-1 会是一个很大的数，用的时候 int n = vector.size()-1 就可以了.
```

☆C++

- 输入

☆面试题

字符串转为整数

将一个字符串转换成一个整数，要求不能使用字符串转换整数的库函数。数值为0或者字符串不是一个合法的数值则返回0。

注意：（1）字符串中可能出现任意符号，出现除 +/- 以外符号时直接输出0.（2）字符串中可能出现 +/- 且仅可能出现在字符串首位。

```
class Solution {
```

```
public:
    bool isvalid(char ch) {
        if(!(ch >= '0' && ch <= '9')) return false;
        return true;
    }
    int StrToInt(string str) {
        int n = str.size();
        int sum = 0, cnt = 1;
        int flag = 1;

        for(int i = n - 1; i >= 0; i--) {
            char ch = str[i];
            if(ch == '+') continue;
            else if(ch == '-') {
                flag = -1;
                continue;
            }
            else if(!isvalid(str[i])) return 0;
            sum += (str[i] - '0') * cnt;
            cnt *= 10;
        }
        return flag * sum;
    }
};
```

两数之和

输入一个递增排序的数组array和一个数字S，在数组中查找两个数，使得他们的和正好是S，如果有多对数字的和等于S，返回任意一组即可，如果无法找出这样的数字，返回一个空数组即可。

```
// 1. 这种做法不对，会重复使用，比如 [1,5,11],10 的样例会返回 [5,5]
// 错在两个循环上，遍历一次就可以了
class Solution {
public:
    vector<int> FindNumbersWithSum(vector<int> array,int sum) {
        unordered_set<int> set;
        int n = array.size();
        if(n < 2) return {};

        for(int i = 0; i < n; i++) {
            set.insert(array[i]);
        }

        for(int i = 0; i < n; i++) {
            auto iter = set.find(sum - array[i]);
            if(iter != set.end()) return {array[i], sum - array[i]};
        }

        return {};
    }
};
// 2.
class Solution {
public:
    vector<int> FindNumbersWithSum(vector<int> array,int sum) {
        unordered_set<int> set;
        int n = array.size();
        if(n < 2) return {};

        for(int i = 0; i < n; i++) {
            auto iter = set.find(sum - array[i]);
            if(iter != set.end()) return {array[i], sum - array[i]};
            else set.insert(array[i]);
        }

        return {};
    }
};
```