华为 001 最佳检测顺序

自底向上遍历

用全局变量

保存父节点 下标

```cpp
//最佳检测顺序
//自底向上遍历

#include<iostream>
#include<algorithm>
using namespace std;

struct node {
    int index = 0;
    int cnt = 0;
};

node ans[100000];
int fa[100000];

bool cmp(node a, node b) {
    if (a.cnt == b.cnt) return a.index < b.index;
    else return a.cnt > b.cnt;
}

void func(int index) {
    if (index == -1) return;

    ans[index].cnt++;
    func(fa[index]);
}

int main() {
    int n;
    cin >> n;
    for (int i = 0; i < n; i++) {
        cin >> fa[i];
        ans[i].index = i;
    }

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

    sort(ans, ans + n, cmp);
    for (int i = 0; i < n; i++) {
        if (i != 0) cout << " ";
        cout << ans[i].index;
    }

    return 0;
```

```
}
```

560 和为k的连续子数组

重要

```
int subarraySum(vector<int>& nums, int k) {
    int ans = 0, cursum = 0;
    unordered_map<int,int> mp;
    mp[0] = 1;
    for(int x : nums) {
        cursum += x;
        if(mp.count(cursum - k)) {
            ans += mp[cursum-k];
        }
        mp[cursum]++;
    }
    return ans;
}
```

rand7 生成 rand10

文章基于这样一个事实 (randX() - 1)*Y + randY() 可以等概率的生成[1, X * Y]范围的随机数

接雨水

左右指针

```
int trap(vector<int>& height) {
    int l=0, r=height.size()-1;
    int lmax = 0, rmax = 0, ans = 0;
    while(l < r) {
        lmax = max(height[l], lmax);
        rmax = max(height[r], rmax);
        if(lmax < rmax) {
            ans += lmax - height[l];
            l++;
        } else {
            ans += rmax - height[r];
            r--;
        }
    }
    return ans;
}
```

小于n的最大数

对数组排序 然后回溯

```
int ans = INT_MIN;
void dfs(vector<int>& nums, int cur, int target, int targetlen, int curlen) {
    if (curlen > targetlen || cur >= target)
        return;
    ans = max(ans, cur);
    for (int i = 0; i < nums.size(); i++) {
        dfs(nums, cur*10 + nums[i], target, targetlen, curlen+1);
        //这里不能 ++ 会导致后面都改变，应该写为 +1
    }
}

int main() {
    int target;
    cin >> target;
    //vector<int> nums = { 1,2,4,9 };
    vector<int> nums = { 2,4,5};
    int len = 0, tmp = target;
    while (tmp) {
        tmp /= 10;
        len++;
    }
    //int len = to_string(target).size();
    sort(nums.begin(), nums.end());
    dfs(nums, 0, target, len, 0);
    cout << ans << endl;
    return 0;
}
```

31 下一个排列 **

先判断是否为递减序列（从大到小），所以结果为反转整个数组

然后从后往前找 到第一个小于 i-1 位置的数据 交换两个位置数据

然后将后面所有数据从小到大排列（即 反转后面所有数据

2 3 5 4 1 -> 2 4 5 3 1 -> 2 4 1 3 5

交换 3 4 然后将后面反转（从小到大

```
void nextPermutation(vector<int>& nums) {
    int i=nums.size()-1;
    while(i-1 >= 0 && nums[i-1] >= nums[i])
        i--;
```

```
    if(i-1 < 0) {
        reverse(nums.begin(), nums.end());
    } else {
        int j=nums.size()-1;
        //找到第一个大于 nums[i-1]的数
        while(i-1 < j && nums[i-1] >= nums[j])
            j--;

        swap(nums[i-1], nums[j]); //交换nums[i-1] nums[j]
        reverse(nums.begin()+i, nums.end());
        //然后将i-1后面的数 从小到大排序
    }
}
```

146 LRU缓存

记住结构体怎么写

只有链表中的节点才需要从list删除，新节点直接加入到head后面

记住往 mp中添加 和删除

```
struct Node {
    int key, value;
    Node *pre, *next;
    Node():key(0),value(0),pre(nullptr),next(nullptr){}
    Node(int key_, int value_):key(key_), value(value_), pre(nullptr),
next(nullptr){}
};

class LRUCache {
    public:
    int size, cap;
    Node *head, *tail;
    unordered_map<int,Node*> mp;
    LRUCache(int capacity) {
        size = 0, cap = capacity;
        head = new Node();
        tail = new Node();
        head->next = tail;
        tail->pre = head;
    }

    int get(int key) {
        if(mp.count(key)) {
            Node *node = mp[key];
            removeFromList(node);
            addToHead(node);
            return node->value;
        }
        return -1;
    }
```

```
    void put(int key, int value) {
        if(mp.count(key)) {
            Node *node = mp[key];
            node->value = value;
            removeFromList(node);
            addToHead(node);
            return;
        }
        Node *node = new Node(key,value);
        addToHead(node);
        mp[key] = node;
        size++;
        if(size > cap) {
            Node *tmp = tail->pre;
            removeFromList(tmp);
            mp.erase(tmp->key);
            size--;
        }
    }

    void removeFromList(Node *node) {
        Node *pre = node->pre;
        Node *next = node->next;
        pre->next = next;
        next->pre = pre;
    }

    void addToHead(Node *node) {
        Node *pre = head;
        Node *next = head->next;
        node->next = next;
        next->pre = node;
        node->pre = pre;
        pre->next = node;
    }
};
```

129 求根节点到叶节点数字之和

```
int ans = 0;
int sumNumbers(TreeNode* root) {
    dfs(root, 0);
    return ans;
}

void dfs(TreeNode* root, int cur) {
    if(!root) return;
    cur = cur * 10 + root->val;
    if(!root->left && !root->right) {
        ans += cur;
    }
    dfs(root->left,cur);
    dfs(root->right,cur);
```

```
}
```

181 买卖股票的最佳时机 IV

你最多可以完成 **k** 笔交易

买和卖 不能在同一天

121 买卖股票的最佳时机 I

dp 动态规划

- dp含义
- 递推公式
- 初始化
- 遍历顺序

这道题 股票只能买卖一次！！！

```
dp[i][0] 持有股票的最大金额
dp[i][1] 不持有股票的最大金额

dp[i][0] = max(dp[i-1][0], -price[i]) //只能买卖一次 所以如果买了 直接是 -price[i]
dp[i][1] = max(dp[i-1][1], dp[i-1][0] + price[i])
```

```
int maxProfit(vector<int>& prices) {
    vector<vector<int>> dp(prices.size(), vector<int>(2,0));
    dp[0][0] = -prices[0]; //0 持有
    dp[0][1] = 0; //1 未持有
    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];
}
```

165 比较版本号

```
int compareVersion(string version1, string version2) {
    int i = 0, m = version1.size();
    int j = 0, n = version2.size();
    while(i < m || j < n) {
```

```
        int a = 0, b = 0;
        while(i<m && version1[i] != '.') {
            a = a * 10 + (version1[i] - '0');
            i++;
        }
        i++;
        while(j<n && version2[j] != '.') {
            b = b * 10 + (version2[j] - '0');
            j++;
        }
        j++;
        if(a != b) return a > b ? 1 : -1;
    }
    return 0;
}
```

708 两数相加

链表两数相加

```
ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
    ListNode* newhead = new ListNode();
    ListNode* l3 = newhead;
    int add = 0;
    while(l1 || l2 || add) {
        int n1 = l1 == nullptr ? 0 : l1->val;
        int n2 = l2 == nullptr ? 0 : l2->val;
        int n3 = (n1 + n2 + add) % 10;
        add = (n1 + n2 + add) / 10;
        ListNode* node = new ListNode(n3);
        l3->next = node;
        l3 = node;
        if(l1) l1 = l1->next;
        if(l2) l2 = l2->next;
    }
    return newhead->next;
}
```

841 钥匙和房间

深度搜索

```
bool canVisitAllRooms(vector<vector<int>>& rooms) {
    vector<bool> vis(rooms.size());
    int num = 0;
    dfs(rooms, vis, num, 0);
    return num == rooms.size();
}
```

```
void dfs(vector<vector<int>>& rooms, vector<bool>& vis, int& num, int index) {
    vis[index] = true;
    num++;
    for(auto it : rooms[index]) {
        if(!vis[it]) {
            dfs(rooms, vis, num, it);
        }
    }
}
```

76 最小覆盖子串

```
string minWindow(string s, string t) {
    unordered_map<char,int> hs, ht;
    string ans;
    for(int i=0; i<t.size(); i++) {
        ht[t[i]]++;
    }
    for(int i=0, j=0, cnt=0; i<s.size(); i++) {
        hs[s[i]]++;
        if(hs[s[i]] <= ht[s[i]]) cnt++; //小于等于
        while(hs[s[j]] > ht[s[j]]) {
            hs[s[j]]--;
            j++;
        }
        if(cnt == t.size()) {
            if(ans.empty() || ans.size() > i-j+1) {
                ans = s.substr(j, i-j+1);
            }
        }
    }
    return ans;
}
```

蚂蚁0316 组装电脑

dfs 由于每个零件都要取一个，所以用深度搜索

```
#include<iostream>
#include<vector>
using namespace std;
struct Node {
    long long price, val;
    Node() :price(0), val(0) {}
};

void dfs(vector<vector<Node>>& vec, long long n, long long x,long long index,
long long curcost, long long val, long long& ans){
    //cout << "dfs:" << index << " " << curcost << " " << val << " " << endl;
```

```
    if (index == n) {
        //index从0开始
        if (curcost <= x)
            ans = max(ans, val);
        return;
    }
    for (long long i = 0; i < (long long)vec[index].size(); i++) {
        if (curcost + vec[index][i].price > x) continue;
        dfs(vec, n, x, index + 1, curcost + vec[index][i].price, val +
vec[index][i].val, ans);
    }
}

int main() {
    long long n, x;
    cin >> n >> x;
    vector<vector<Node>> vec;
    for (long long i = 0; i < n; i++) {
        long long m;
        cin >> m;
        vector<Node> tmp(m);
        for (long long j = 0; j < m; j++)
            cin >> tmp[j].price;
        for (long long j = 0; j < m; j++)
            cin >> tmp[j].val;
        vec.push_back(tmp);
    }
    /*for (int i = 0; i < n; i++) {
        for (int j = 0; j < vec[i].size(); j++)
            cout << vec[i][j].price << "," << vec[i][j].val << " ";
        cout << endl;
    }*/

    long long ans = -1;
    dfs(vec, n, x, 0, 0, 0, ans);
    cout << ans;
    return 0;
}
```

28 找出字符串中第一个匹配项的下标

时间复杂度 m*n

```
int strStr(string haystack, string needle) {
    int m=haystack.size(), n=needle.size();
    if(m<n) return -1;
    for(int i=0; i<=m-n; i++) {
        int k=i, j=0; // k j 初始化为0
        for(j=0;j<n;k++,j++) {
            if(haystack[k] != needle[j])
                break; //不相等 则退出
```

```
        }
        if(j==n)
            return i;
    }
    return -1;
}
```

KMP算法 时间复杂度 m+n

next数组 存储字符串中每个字符 对应的最长前后缀匹配长度

https://www.bilibili.com/video/BV1PD4y1o7nd/?spm_id_from=333.999.0.0

万万没想到之抓捕孔连顺

确定i的位置，从前面 j ~ i-1 选2个数

```
long func(long n) {
    return n*(n-1)/2;
}

int main() {
    int N, D;
    cin>>N>>D;
    long ans = 0;
    int j=0;
    vector<int> vec(N,0);
    for(int i=0; i<N; i++) {
        cin>>vec[i];
        while(i>=2 && vec[i]-vec[j]>D) {
            j++;
        }
        ans += func(i-j);
    }


    cout<<ans % 99997867 ;
}
```

万万没想到之聪明的编辑

三个同样的字母连在一起， 去掉一个的就好啦：比如 helllo -> hello

两对一样的字母（AABB型）连在一起， 去掉第二对的一个字母就好啦：比如 helloo -> hello

```
string func(string str) {
    for (int i = 0; i < str.size(); ) {
        int n = str.size();
        if (i + 3 < n && str[i] == str[i + 1] && str[i + 2] == str[i + 3]) {
            str.erase(i+3,1);
```

```
        } else if (i + 2 < n && str[i] == str[i + 1] && str[i] == str[i + 2]) {
            str.erase(i+2,1);
        } else {
            i++;
        }

    }
    return str;
}
```

字符串相减

415 字符串相加

```
string addStrings(string num1, string num2) {
    int i=num1.size()-1, j=num2.size()-1, add=0;
    string ans = "";
    while(i>=0 || j>=0 || add != 0) {
        int x = i>=0 ? num1[i]-'0' : 0;
        int y = j>=0 ? num2[j]-'0' : 0;
        int res = x + y + add;
        ans.push_back('0' + res % 10);
        add = res / 10;
        i--, j--;
    }
    reverse(ans.begin(), ans.end());
    return ans;
}
```

1109 航班预定统计

209 长度最小的子数组

```
int minSubArrayLen(int target, vector<int>& nums) {
    int ans = INT_MAX;
    int i=0, j=0, cur=0;
    for(j=0; j<nums.size(); j++) {
        cur += nums[j];
        while(cur>=target) {
            ans = min(ans, j-i+1);
            cur -= nums[i];
            i++;
        }
    }
    return ans == INT_MAX ? 0 : ans;
}
```

958 二叉树的完全性检验

对于一个完全二叉树，层序遍历的过程中**遇到第一个空节点之后不应该再出现非空节点**

```
bool isCompleteTree(TreeNode* root) {
    queue<TreeNode*> que;
    que.push(root);
    bool flag = false;
    while(!que.empty()) {
        TreeNode *node = que.front();
        que.pop();
        if(!node) {
            flag = true;
            continue;
        }
        if(flag) return false; //前面有了空节点情况 又出现了非空节点
        que.push(node->left);
        que.push(node->right);//不管是否为空节点 都会放入队列
    }
    return true;
}
```

94 二叉树的中序遍历

非递归 使用栈

```
vector<int> inorderTraversal(TreeNode* root) {
    stack<TreeNode*> st;
    vector<int> ans;
    TreeNode* cur = root;
    while(cur || !st.empty()) {
```

```
        while(cur) {
            st.push(cur);
            cur = cur->left;
        }
        cur = st.top();
        st.pop();
        ans.emplace_back(cur->val);
        cur = cur->right;
    }
    return ans;
}
```

序列和

给出一个正整数N和长度L，找出一段长度大于等于L的**连续非负整数**，他们的和恰好为N；找出长度最小的那个

从小到大输出这段连续非负整数，以空格分隔，**行末无空格**

用等差数列公式

```
//Sn = (a1 + an) * n / 2;
//即 2N = [a1 + (n-1)*d + a1] * n
//其中d = 1
//2N = (2a1 + n - 1) * n
// a1 = (2N+n-n*n) / 2n
```

从L 到 100 找到满足条件的a1 则输出

尝试所有的n n就是长度，所以从小到大遍历

```
#include <iostream>
using namespace std;

int main() {
    int N, L;
    while (cin >> N >> L) { // 注意 while 处理多个 case
    bool flag = false;
        for(int i=L; i<=100; i++) {
            if( (2*N+i-i*i) % (2*i)== 0 && (2*N+i-i*i)>= 0) {
                int a1 = (2*N+i-i*i) / (2*i);
                int j=0;
                for(j=0; j < i-1; j++)
                    cout<<a1 + j << " ";
                cout<<a1 + j;
                flag = true;
                break;
            }
        }
        if(!flag)
            cout<<"No";
    }
}
```

```
// 64 位输出请用 printf("%lld")
```

1143 最长公共子序列

dp[i] [j] 代表 text1 从0 - i-1 和 text2 从 0 - j-1 的最长公共子序列

```
int longestCommonSubsequence(string text1, string text2) {
    int m = text1.size(), n = text2.size();
    vector<vector<int>> dp(m+1, vector<int>(n+1, 0));
    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];
}
```

23 合并K个升序链表

使用**优先队列（堆）来辅助排序节点**，排序时间复杂度 longN

```
struct Cmp {
    bool operator() (ListNode* a, ListNode* b) {
        return a->val > b->val;
    }// 大根堆
};
ListNode* mergeKLists(vector<ListNode*>& lists) {
    priority_queue<ListNode*, vector<ListNode*>,Cmp> que;
    ListNode *newhead = new ListNode(), *cur = newhead;
    for(int i=0; i<lists.size(); i++) {
        if(lists[i])
            que.push(lists[i]);
    }
    while(!que.empty()) {
        ListNode* top = que.top();
        que.pop();
        cur->next = top;
        cur = cur->next;
        if(top->next) que.push(top->next);
    }
    return newhead->next;
}
```

138 复制带随机指针的链表

```
Node* copyRandomList(Node* head) {
    unordered_map<Node*,Node*> mp;
    Node *cur = head;
    for(cur = head; cur; cur = cur->next) {
        mp[cur] = new Node(cur->val);
    }
    for(cur = head; cur; cur = cur->next) {
        if(cur->next) mp[cur]->next = mp[cur->next];
        if(cur->random) mp[cur]->random = mp[cur->random];
    }
    return mp[head];
}
```

72 编辑距离

三种操作：插入一个字符、删除一个字符、替换一个字符

f(i, j) 表示 s1的前i个字符 和 s2的前 j 个字符 匹配的 最少操作数

10 正则表达式

动态规划

f(i,j) s的前i个字符 和 p的前j 个字符匹配

- p[j] != *

  f(i,j) = f(i-1, j-1) && (i, j)

- p[j] = *

  f(i,j) = f(i, j-2) || f(i-1, j) && (i, j-1)

168 Excel表列名称

从 1 开始的的 2626 进制转换题

需要从 1 开始，因此在执行「进制转换」操作前，我们需要先对 columnNumber 执行减一操作，从而实现整体偏移

```
string convertToTitle(int columnNumber) {
    string ans;
    while(columnNumber > 0) {
        columnNumber--;
        ans.push_back(columnNumber % 26 + 'A');
        columnNumber /= 26;
    }
    reverse(ans.begin(), ans.end());
    return ans;
}
```

679 24点游戏

C/C++中double类型除以0会得到inf, 不会报错

```
bool judgePoint24(vector<int>& cards) {
    vector<double> nums;
    for(int x : cards) {
        nums.push_back(double(x));
    }
    return dfs(nums);
}

bool dfs(vector<double>& nums) {
    if(nums.size() == 1) {
        return fabs(nums[0]-24)<=(1e-6);
    }
    for(int i=0; i<nums.size(); i++) {
        for(int j=0; j<nums.size(); j++) {
            if(i==j) continue;
            double a=nums[i];
            double b=nums[j];
            vector<double> res;
            for(int k=0; k<nums.size(); k++) {
                if(k==i || k==j) continue;
                res.push_back(nums[k]);
            }
            vector<double> vec;
            vec.push_back(a+b);
            vec.push_back(a-b);
            vec.push_back(a*b);
            if(fabs(b)>(1e-6))
                vec.push_back(a/b); //不为0 则可以除

            for(int w=0; w<vec.size(); w++) {
                res.push_back(vec[w]);
                if(dfs(res))
                    return true;
                res.pop_back();
            }
        }
    }
    return false;
```

```
}
```

1438 绝对差不超过限制的最长连续子数组

multiset底层是红黑树

通过插入 删除，能够自动排序，便于本题 滑动窗口

注意 删除时候要使用指针 表示位置，如果用val 会删除所有重复值

先插入数据，再判断while

```
int longestSubarray(vector<int>& nums, int limit) {
    multiset<int> st;
    int left = 0, right = 0, n = nums.size(), ans = 0;
    while(right < n) {
        st.insert(nums[right]);
        while(*st.rbegin() - * st.begin() > limit) {
            st.erase(st.find(nums[left]));
            // st.erase(*st.find(nums[left]));  //multiset会将所有重复的元素都删除
            left++;
        }
        ans = max(ans, right - left + 1);
        right++;
    }
    return ans;
}
```

3. 无重复字符的最长子串

和上一题解题思路很类似

这道题 要先判断while 再插入数据

```
int lengthOfLongestSubstring(string s) {
    int ans = 0, left = 0;
    unordered_set<char> st;
    for(int i=0; i<s.size(); i++) {
        while(st.count(s[i])) {
            st.erase(s[left]);
            left++;
        }
        st.insert(s[i]);
        ans = max(ans, i-left+1);
    }
    return ans;
}
```

718 最长重复子数组

动态规划

dp[i] [j] 以i-1结尾的nums1 和以 j-1结尾的nums2 的最大重复子数组长度

如果dp[i] [j]标识以i j为结尾的位置，则需要单独处理dp数组的 第0行 和第0列

递推公式

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

```
int findLength(vector<int>& nums1, vector<int>& nums2) {
    int m=nums1.size(), n=nums2.size(), ans=0;
    vector<vector<int>> dp(m+1, vector<int>(n+1,0));
    for(int i=1; i<=m; i++) {
        for(int j=1; j<=n; j++) {
            if(nums1[i-1] == nums2[j-1])
                dp[i][j] = dp[i-1][j-1] + 1;
            ans = max(ans, dp[i][j]);
        }
    }
    return ans;
}
```

560 和为k的子数组

统计并返回 该数组中和为 `k` 的连续子数组的个数

```
int subarraySum(vector<int>& nums, int k) {
    unordered_map<int, int> mp;// key是前缀和 val有几个符合这个前缀和
    mp[0]=1;
    int cursum = 0, ans = 0;
    for(int x : nums) {
        cursum += x;
        if(mp.count(cursum-k)) {
            ans += mp[cursum-k];
        }
        mp[cursum]++;
    }
    return ans;
}
```

最小栈

用辅助栈 存储当前状态对应的最小值

```
class MinStack {
    stack<int> minstack;
    stack<int> stk;
    public:
    MinStack() {
        minstack.push(INT_MAX);
    }

    void push(int val) {
        stk.push(val);
        minstack.push(min(minstack.top(), val));
    }

    void pop() {
        stk.pop();
        minstack.pop();
    }

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

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

双栈排序

```
stack<int> stackSort(stack<int> &stk) {
    stack<int> tmp;
    while(!stk.empty()) {
        int val = stk.top();
        stk.pop();
        while(!tmp.empty() && tmp.top()>val) {
            int t = tmp.top();
            tmp.pop();
            stk.push(t);
        }
        tmp.push(val);
    }
    return tmp;
}
```

19 删除链表的倒数第 N 个结点

```
ListNode* removeNthFromEnd(ListNode* head, int n) {
    ListNode *newhead = new ListNode(0,head);
```

```
    ListNode *slow = newhead, *fast = newhead;
    while(n--) {
        fast = fast->next;
        if(fast == nullptr) return nullptr;
    }
    while(fast->next) {
        slow = slow->next;
        fast = fast->next;
    }
    slow->next = slow->next->next;
    return newhead->next;
}
```

494 目标和

两部分集合 P N

> sum(P) + sum(N) = sum
>
> sum(P) - sum(N) = S
>
> 得到 sum(P) = (sum + S) / 2

如果 S 大于 sum 不可能实现； 如果sum + S 不是偶数，也不对

dp[j] 装满容量为j 的背包， 有dp[j]种方法

转移方程 dp[j] = dp[j] + dp[j - num[i]] 问有多少种方法， 需要累加

dp[0] 设置为 1

第二个for循环是倒着遍历

```
int findTargetSumWays(vector<int>& nums, int target) {
    int sum = 0;
    for(int i=0; i<nums.size(); i++)
        sum += nums[i];
    if(target > sum) return 0;
    if((sum + target) % 2 == 1) return 0;

    int cap = (sum + target) / 2;
    if(cap < 0) return 0;
    vector<int> dp(cap+1, 0);
    dp[0] = 1;
    for(int i=0; i<nums.size(); i++) {
        //每次取一个数 nums[i] 影响方法数 是累加
        for(int j=cap; j>=nums[i]; j--) {
            //这里是倒着遍历 因为每个数要么取 要么不取，不是无限个数
            dp[j] += dp[j-nums[i]];
        }
    }
    return dp[cap];
}
```

518 零钱兑换II

dp 动态规划 完全背包

递推公式和 上一题一样 dp[j] += dp[j -nums[i]]   dp[0] = 1

完全背包，由于每个物品能够被添加多次，所以要从小到大遍历背包容量（第二个for循环）

```
int change(int amount, vector<int>& coins) {
        vector<int> dp(amount+1,0);
        dp[0]=1;
        for(int i=0; i<coins.size(); i++) {
            for(int j=coins[i]; j<=amount; j++) {
                //完全背包 每个物品无限个 所以要从小到大遍历
                dp[j] += dp[j-coins[i]];
            }
        }
        return dp[amount];
    }
```

322 零钱兑换

dp 动态规划 完全背包

本题求钱币最小个数，那么钱币有顺序和没有顺序都可以，都不影响钱币的最小个数

五部曲

- dp[j] 装满容量为j的背包 最少物品为dp[j]
- 递推公式 dp[j] = min(dp[j- coins[i]] + 1, dp[j]);
- 初始值 dp[0] = 0 其他INT_MAX
- 遍历顺序

  518 在零钱兑换II 求组合数，所以先遍历物品 再遍历背包

  在组合总数IV 求排列数 所以先遍历背包 再遍历物品

  本题求最小数量，先物品 还是先背包 都可以
- 打印dp数组

先遍历背包

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

先遍历物品

这里j 直接从coins[i] 开始即可

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

139 单词拆分

dp 动态规划

字符串最前面加上 空格

```
bool wordBreak(string s, vector<string>& wordDict) {
    int n = s.size();
    vector<bool> dp(n+1,false);
    unordered_set<string> st(wordDict.begin(), wordDict.end());
    s = ' ' + s;
    dp[0]=true;
    //得加上这个空格 并设置true  不然if判断第一个条件会遇到特殊处理
    for(int i=1; i<=n; i++) {
```

```
        for(int j=i; j-1 >= 0; j--) {
            if(dp[j-1] && st.count(s.substr(j, i-j+1))) {
                dp[i] = true;
                break;
            }
        }
    }
    return dp[n];
}
```

LRU缓存

双向链表 + map

```
struct Node {
    int key, val;
    Node *pre, *next;
    Node():key(0), val(0), pre(nullptr), next(nullptr){}
    Node(int _key, int _val):key(_key), val(_val), pre(nullptr), next(nullptr)
{}
};

class LRUCache {
    public:
    Node *head, *tail;
    unordered_map<int, Node*> mp;
    int cap, size;

    LRUCache(int capacity) {
        cap = capacity, size = 0;
        head = new Node();
        tail = new Node();
        head->next = tail;
        tail->pre = head;
    }

    int get(int key) {
        if(mp.count(key) == 0) return -1;
        Node *node = mp[key];
        remove(node);
        addToHead(node);
        return node->val;
    }

    void put(int key, int value) {
        if(mp.count(key) != 0) {
            Node *node = mp[key];
            node->val = value;
            remove(node);
            addToHead(node);
            return ;
        }
        Node *node = new Node(key,value);
```

```
        mp[key] = node;
        addToHead(node);
        size++;
        if(size > cap) {
            Node *tmp = tail->pre;
            mp.erase(tmp->key);
            remove(tmp);
            size--;
            delete tmp;
        }
    }

    //这个函数只针对在链当中的才可以
    void remove(Node *node) {
        node->pre->next = node->next;
        node->next->pre = node->pre;
    }

    void addToHead(Node *node) {
        node->next = head->next;
        head->next->pre = node;
        node->pre = head;
        head->next = node;

    }
};
```

41 缺失的第一个正数

遍历两次数组，然后不断交换位置

```
//第i个位置 存放 数字i+1
//数字i 存放在 i-1位置
//即 nums[i] == nums[nums[i]-1]
int firstMissingPositive(vector<int>& nums) {
    int n=nums.size();
    for(int i=0; i<n; i++) {
        while(nums[i]>=1 && nums[i]<=n && nums[i]!=nums[nums[i]-1]) {
            swap(nums[i], nums[nums[i]-1]);
        }
    }
    for(int i=0; i<n; i++) {
        if(nums[i] != i+1) return i+1;
    }
    return n+1;
}
```

148 排序链表

数据量为50000 不能用插入排序

要求时间复杂度 O(n log n) 归并排序 由于要求空间复杂度常数，所以不能自顶向下，要自低向上

147 对链表进行插入排序

时间复杂度O(n^2) 数据量为5000

找到第一个大于待插入的节点，将其插入到其前面

相当于两条链表 将待排序的节点依次插入到另一条已排序的链表

```
ListNode* insertionSortList(ListNode* head) {
    ListNode *newhead = new ListNode(0), *cur = head;
    while(cur) {
        ListNode *pre = newhead, *next = cur->next;
        while(pre->next && pre->next->val <= cur->val)
            pre = pre->next;
        cur->next = pre->next;
        pre->next = cur;
        cur = next;
    }
    return newhead->next;
}
```

153 寻找旋转排序数组中的最小值

这道题，判断中间元素和右边界即可：如果nums[mid] > nums[r] 则l-mid一定是升序，且min在mid+1 - r

由于while循环 l<r 所以mid绝不会等于r

如果nums[mid]<nums[r] 则最小值在l - mid，因为mid-1位置可能会是最大值

当 nums[mid] < nums[r] 中间元素小于最右边元素判断；则最小元素的区间在 l ~ mid

当nums[mid] > nums[r]； 说明mid右边有转折区间 则最小元素区间在 mid+1 ~ r

只要区间元素大于1 mid就不会等于high

```
比如 0 1 , mid=(0+1)/2=0
比如 5 6 , mid=(5+6)/2=5
```

```
int findMin(vector<int>& nums) {
    int l=0, r=nums.size()-1;
    while(l < r) {
        int mid=l+(r-l)/2;
        if(nums[mid] < nums[r]) r=mid;
        else l=mid+1;
    }
    return nums[l];
}
```

69 x的平方根

注意和上一题 在求 mid 的区别

```
int mySqrt(int x) {
    if(x == 0) return 0;
    int l=1, r=x;
    while(l < r) {
        int mid = l + (r - l) / 2 + 1;
        //由于向下取整 mid可能取到l 而l还可能取mid 导致死循环 所以要加1
        if(mid <= x / mid) l = mid;
        else r = mid - 1;
    }
    return l;
}
```

```
int mySqrt(int x) {
    // if(x == 0) return 0;
    int l=0, r=x;
    while(l < r){
        int mid = l + (r - l) / 2 + 1;
        if(mid <= x / mid) l = mid;
        else r = mid-1;
    }
    return l;
}
```

剑指offer 09 用两个栈实现队列

初始创建变量 st1 st2

st2的栈顶元素就是正序的 队列元素

注意当st2空 且st1空时 返回-1

```
class CQueue {
public:
    stack<int> st1, st2;
```

```
    CQueue() {

    }

    void appendTail(int value) {
        st1.push(value);
    }

    int deleteHead() {
        int res = 0;
        if(!st2.empty()) {
            res = st2.top();
            st2.pop();
            return res;
        }

        if(st1.empty()) {
            return -1;
        }

        while(!st1.empty()) {
            int value = st1.top();
            st1.pop();
            st2.push(value);
        }
        res = st2.top();
        st2.pop();
        return res;
    }
};
```

169 多数元素

数元素是指在数组中出现次数 **大于** `⌊ n/2 ⌋` 的元素、

```
int majorityElement(vector<int>& nums) {
    int ans=0, cnt=0;
    for(int i=0; i<nums.size(); i++) {
        if(cnt == 0) {
            cnt=1;
            ans=nums[i];
            continue;
        }
        if(ans == nums[i]) cnt++;
        else cnt--;
    }
    return ans;
}
```

关于 除2

奇数个情况：如5

- 5/2=2 （下标 0 1 2 3 4 下标2 刚好是中间

偶数个情况：如6

- 6/2=3 （下标 0 1 2 3 4 5 下标2 3 是中间；for循环判断用pre now 保存

4 寻找两个正序数组的中位数

注意点

- 返回结果 偶数情况

  ```
  double((pre+now)/2.0);
  ```

- 判断条件

  注意数组越界情况

  ```
  //判断bstart要写在nums2[bstart]前面 防止越界
  astart < m && (bstart >= n || (nums1[astart] <= num2[bstart]))
  ```

```
double findMedianSortedArrays(vector<int>& nums1, vector<int>& nums2) {
    int m=nums1.size(), n=nums2.size(), len=m+n;
    int pre=0, now=0, astart=0, bstart=0;
    for(int i=0; i<=len/2; i++) {
        pre = now;
        if(astart<m && (bstart>=n || nums1[astart]<=nums2[bstart]))
            now=nums1[astart++];
        else
            now=nums2[bstart++];
    }
    if(len & 1 == 1) return now;
    else return double((pre+now)/2.0);
}
```

22 括号生成

用回溯

判断条件

- cur.size() == n*2;
- left < n; 深度搜索到全是左括号，然后开始添加右括号
- right < left;

```
vector<string> generateParenthesis(int n) {
    vector<string> ans;
    string cur;
    backtrack(ans, cur, 0, 0, n);
    return ans;
}

void backtrack(vector<string>& ans, string& cur, int left, int right, int n) {
    if(cur.size() == n*2) {
        ans.push_back(cur);
        return;
    }

    if(left < n) {
        cur.push_back('(');
        backtrack(ans, cur, left+1, right, n);
        cur.pop_back();
    }
    if(right < left ) {
        cur.push_back(')');
        backtrack(ans, cur, left, right+1, n);
        cur.pop_back();
    }
}
```

450 删除二叉搜索树中的节点

根据二叉搜索树的性质

如果目标节点大于当前节点值，则去右子树中删除；
如果目标节点小于当前节点值，则去左子树中删除；
如果目标节点就是当前节点，分为以下三种情况：
其无左子：其右子顶替其位置，删除了该节点；
其无右子：其左子顶替其位置，删除了该节点；
其左右子节点都有：其左子树转移到其右子树的最左节点的左子树上，然后右子树顶替其位置，由此删除了该节点

```
class Solution {
public:
    TreeNode* deleteNode(TreeNode* root, int key)
    {
        if (root == nullptr)    return nullptr;
        if (key > root->val)
            root->right = deleteNode(root->right, key); // 去右子树删除
        else if (key < root->val)
            root->left = deleteNode(root->left, key);  // 去左子树删除
        else    // 当前节点就是要删除的节点
        {
            if (! root->left)   return root->right; // 情况1，欲删除节点无左子
            if (! root->right)  return root->left;  // 情况2，欲删除节点无右子
            TreeNode* node = root->right;           // 情况3，欲删除节点左右子都有
            while (node->left)          // 寻找欲删除节点右子树的最左节点
```

```
                node = node->left;
            node->left = root->left;    // 将欲删除节点的左子树成为其右子树的最左节点的
左子树
            root = root->right;         // 欲删除节点的右子顶替其位置，节点被删除
        }
        return root;  //最后返回root
    }
};
```

124 二叉树中的最大路径和

后序遍历：先计算子节点，然后在根节点计算

dfs函数：该节点为根节点的子树中寻找**以该节点为起点**的一条路径

- 该函数不能直接计算出最终结果，只是得到一部分，最终结果保存在全局变量
- 并且只保留其中 正的部分

```
int ans = INT_MIN;
int maxPathSum(TreeNode* root) {
    dfs(root);
    return ans;
}

int dfs(TreeNode* root) {
    if(!root) return 0;
    int left = max(0, dfs(root->left));
    int right = max(0, dfs(root->right));

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

}
```

堆排序 912

从中间开始 数组长度n，最后一个位置为n-1，**中间的下标为[(n-1)-1] / 2, 即 n/2 - 1**

```
//大顶堆
//堆排序 会将堆顶元素和最后一个元素交换
//所以大顶堆是 升序排列
//小顶堆是 降序排列

//参数n是长度 取不到的边界
void heapsort(vector<int>& nums, int n) {
    //建堆
    for(int i = n/2 - 1; i>=0; i--)
        heapify(nums,n,i);
```

```
    //排序
    for(int i = n-1; i>=0; i--) {
        swap(nums[i], nums[0]);
        heapify(nums,i,0);//移除一个元素后 当前总数为i
    }
}

// nums 存储堆的数组
// n 数组长度
// i 待维护节点的下标
void heapify(vector<int>& nums, int n, int i) {
    //下标从0开始 所以左节点 i*2+1, 右节点 i*2+2;
    int index = i;
    int l=i*2+1, r=i*2+2;

    //n是长度 l r i 是下标 所以判断条件是 小于
    //大顶堆 所以找最大的值
    if(l < n && nums[index] < nums[l]) index=l;
    if(r < n && nums[index] < nums[r]) index=r;
    if(index != i) {
        swap(nums[index], nums[i]);
        heapify(nums, n, index);//递归处理index和其子节点
    }
}
```

归并排序 912

两个函数都有取 mid

划分时候 区间分别为 **l - mid 和 mid+1 - r**

```
void mergesort(vector<int>& nums, int l, int r, vector<int>& ans) {
    if(l >= r) return;
    int mid = (l + r) / 2;
    mergesort(nums, l, mid, ans);
    mergesort(nums, mid+1, r, ans);
    merge(nums,l, r, ans);
}

//参数 r 右边界 能够取到
void merge(vector<int>& nums, int l, int r, vector<int>& ans) {
    if(l>=r) return;
    int mid = (l+r)/2;
    int i=l, j=mid+1, k=0;
    while(i<=mid && j<=r) {
        if(nums[i]<nums[j]) ans[k++] = nums[i++];
        else ans[k++] = nums[j++];
    }
    while(i<=mid) ans[k++] = nums[i++];
    while(j<=r) ans[k++] = nums[j++];

    for(int i=l, k=0; i<=r; i++, k++) {
        nums[i]=ans[k];
```

```
    }
}
```

快排 912

快排 while循环中判断 **i < j**

while结束后 还要 **swap(nums[i], nums[l])**

递归下次的区间为 **l, i-1 和 i+1, r 中间的数已经放在对应位置**

```
//l 左边界 r 右边界 都能取到
void quickSort(vector<int>& nums, int l, int r) {
    if(l >= r) return;
    int i=l, j = r;
    swap(nums[(i+j)/2], nums[l]);
    while(i < j) {
        while(i < j && nums[j] >= nums[l]) j--;
        while(i < j && nums[i] <= nums[l]) i++;
        swap(nums[i], nums[j]);
    }
    swap(nums[i], nums[l]);
    quicksort(nums, l, i-1);
    quicksort(nums, i+1, r);
}
```

```
void quicksort(vector<int>& nums, int l, int r) {
    if(l < r){
        int pos = partition(nums,l,r);
        quicksort(nums,l,pos-1);
        quicksort(nums,pos+1,r);
    }
}

int partition(vector<int>& nums, int l, int r) {
    int i=l, j=r;
    swap(nums[(i+j)/2], nums[l]);
    while(i < j) {
        while(i < j && nums[j] >= nums[l]) j--;
        while(i < j && nums[i] <= nums[l]) i++;
        swap(nums[i], nums[j]);
    }
    swap(nums[i], nums[l]);
    return i;
}
```

215 数组中的第K个最大元素

```
int findKthLargest(vector<int>& nums, int k) {
    int l = 0, r = nums.size()-1, pos = nums.size() - k;
    while(l < r) {
        int mid = quickselect(nums, l, r);
```

```
        if(mid == pos) return nums[mid];
        else if(mid > pos) r = mid - 1;
        else l = mid + 1;
    }
    return nums[l];
}

int quickselect(vector<int>& nums, int l, int r){
    int i=l, j=r;
    while(i<j){
        while(i<j && nums[j]>=nums[l]) j--;
        while(i<j && nums[i]<=nums[l]) i++;
        swap(nums[i], nums[j]);
    }
    swap(nums[i], nums[l]);
    return i;
}
```

141 环形链表

判断是否有环

使用快慢指针

```
class Solution {
public:
    bool hasCycle(ListNode *head) {
        ListNode *slow = head, *fast = head;
        while(fast && fast->next) {
            slow = slow->next;
            fast = fast->next->next;
            if(slow == fast) return true;
        }
        return false;
    }
};
```

142 环形链表II

如果有环，返回入环的第一个节点； 否则返回null

相遇之后确定有环，然后将slow从head开始，两指针每次移动一步，直到相遇

```
ListNode *detectCycle(ListNode *head) {
    ListNode *slow = head, *fast = head;
    while(fast && fast->next) {
        slow = slow->next;
        fast = fast->next->next;
        if(slow == fast) {
            slow = head;
            while(slow != fast) {
```

```
                slow = slow->next;
                fast = fast->next;
            }
            return slow;
        }
    }
    return NULL;
}
```

143 重排链表

注意 fast->next->next 和 判断环形链表的地方

```
L0 → Ln → L1 → Ln - 1 → L2 → Ln - 2 → …
```

1.寻找中间节点 **当快指针后面有两个节点时；就向后移动；**

当奇数个时，返回中间节点； 偶数个时，返回中间节点的前一个节点

```
//判断fast后面是否有两个节点，然后再移动
while(fast->next && fast->next->next) {
    slow = slow->next;
    fast = fast->next->next;
}
```

2.反转后半部分节点

3.插入到前面节点

```
void reorderList(ListNode* head) {
    ListNode *mid = getMidNode(head); //获得中间节点
    ListNode *next = reverseList(mid->next);//将后半部分反转
    mid->next = nullptr; //防止合并后成环
    mergeList(head, next);
}

void mergeList(ListNode* l1, ListNode* l2) {
    while(l1 && l2) {
        ListNode *n1 = l1->next, *n2 = l2->next;
        l1->next = l2, l2->next = n1;
        l1 = n1, l2 = n2;
    }
}

ListNode* getMidNode(ListNode* head) {
    ListNode *slow = head, *fast = head;
    //判断fast后面是否有两个节点，然后再移动
    while(fast->next && fast->next->next) {
        slow = slow->next;
        fast = fast->next->next;
    }
    //如果是奇数个节点 则返回中间节点
```

```
    //偶数个节点 返回中间左侧节点
    return slow;
}

//将链表反转
ListNode* reverseList(ListNode* head) {
    ListNode *pre = new ListNode(0), *cur = head;
    while(cur) {
        ListNode *next = cur->next;
        cur->next = pre->next;
        pre->next = cur;
        cur = next;
    }
    return pre->next;
}
```

21 合并两个有序链表

递归

```
class Solution {
public:
    ListNode* mergeTwoLists(ListNode* list1, ListNode* list2) {
        if(list1 == nullptr) return list2;
        else if(list2 == nullptr) return list1;
        else if(list1->val < list2->val) {
            list1->next = mergeTwoLists(list1->next, list2);
            return list1;
        } else {
            list2->next = mergeTwoLists(list1, list2->next);
            return list2;
        }

    }
};
```

912 排序数组

快排

```
void quickSort(vector<int>& nums, int l, int r) {
    if(l >= r) return;
    int i=l, j = r;
    swap(nums[(i+j)/2], nums[l]);
    while(i < j) {
        while(i < j && nums[j] >= nums[l]) j--;
        while(i < j && nums[i] <= nums[l]) i++;
        swap(nums[i], nums[j]);
    }
    swap(nums[i], nums[l]);
    quicksort(nums, l, i-1);
    quicksort(nums, i-1, r);
}
```

```
void quicksort(vector<int>& nums, int l, int r) {
    if(l < r){
        int pos = partition(nums,l,r);
        quicksort(nums,l,pos-1);
        quicksort(nums,pos+1,r);
    }
}

int partition(vector<int>& nums, int l, int r) {
    int i=l, j=r;
    swap(nums[(i+j)/2], nums[l]);
    while(i < j) {
        while(i < j && nums[j] >= nums[l]) j--;
        while(i < j && nums[i] <= nums[l]) i++;
        swap(nums[i], nums[j]);
    }
    swap(nums[i], nums[l]);
    return i;
}
```

88 合并两个有序数组

从后往前遍历两个数组

```
void merge(vector<int>& nums1, int m, vector<int>& nums2, int n) {
    int index = m + n - 1, i = m - 1, j = n - 1;
    while(i >= 0 && j >= 0) {
        if(nums1[i] >= nums2[j]) nums1[index--] = nums1[i--];
        else nums1[index--] = nums2[j--];
    }

    while(j >= 0) nums1[index--] = nums2[j--];
}
```

20 有效的括号

用栈模拟

注意 直接是右括号时候 判断sta是否为空

```
bool isValid(string s) {
    stack<char> sta;
    for(int i=0; i<s.size(); i++) {
        char ch = s[i];
        if(ch == '(' || ch == '{' || ch == '[')
            sta.push(ch);
        else {
            if(sta.empty()) return false;
            char top = sta.top();
            sta.pop();
            if(top == '(' && ch == ')') continue;
            else if(top == '{' && ch == '}') continue;
            else if(top == '[' && ch == ']') continue;
            else return false;
        }
    }
    return sta.empty();
}
```

674 最长连续递增序列

只用遍历一遍

```
int findLengthOfLCIS(vector<int>& nums) {
    vector<int> dp(nums.size(), 1);
    int ans = 1;
    for(int i=1; i<nums.size(); i++) {
        if(nums[i]>nums[i-1])
            dp[i] = dp[i-1] + 1;
        ans = max(ans, dp[i]);
    }
    return ans;
}
```

300 最长递增子序列

dp 两层for循环

```
int lengthOfLIS(vector<int>& nums) {
    vector<int> dp(nums.size(), 0);
```

```
    int ans = 0;
    for(int i=0; i<nums.size(); i++) {
        dp[i] = 1;
        for(int j=0; j<i; j++) {
            if(nums[j] < nums[i]) {
                dp[i] = max(dp[i], dp[j]+1);
            }
        }
        ans = max(ans, dp[i]);
    }
    return ans;
}
```

46 全排列

回溯 树形结构！！！

使用used数组

```
vector<vector<int>> ans;
vector<int> cur;
vector<vector<int>> permute(vector<int>& nums) {
    vector<bool> visited(nums.size(), false);
    backtrack(nums,visited);
    return ans;
}

void backtrack(vector<int>& nums, vector<bool>& visited) {
    if(cur.size() == nums.size()) {
        ans.push_back(cur);
        return;
    }

    for(int i=0; i<nums.size(); i++) {
        if(visited[i]) continue;
        visited[i] = true;
        cur.push_back(nums[i]);
        backtrack(nums, visited);
        cur.pop_back();
        visited[i] = false;
    }
}
```

47 全排列II

加上排序 和 去重即可

去重 同一层去重判断 used[i-1] = false

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

```
vector<vector<int>> ans;
vector<int> cur;
vector<vector<int>> permuteUnique(vector<int>& nums) {
    sort(nums.begin(), nums.end());
    vector<bool> visited(nums.size(), false);
    backtrack(nums,visited);
    return ans;
}

void backtrack(vector<int>& nums, vector<bool>& visited) {
    if(cur.size() == nums.size()) {
        ans.push_back(cur);
        return;
    }

    for(int i=0; i<nums.size(); i++) {
        if(visited[i]) continue;
        if(i-1 >= 0 && nums[i-1] == nums[i] && visited[i-1] == false) continue;
        visited[i] = true;
        cur.push_back(nums[i]);
        backtrack(nums,visited);
        cur.pop_back();
        visited[i]=false;
    }
}
```

5 最长回文子串

参数使用引用 计算结果判断是否更新

最后left right 是不满足的位置，所以要-2

```
string longestPalindrome(string s) {
    string res = "";
    for(int i=0; i<s.size(); i++) {
        extendFromCenter(s, res, i,i);
        extendFromCenter(s, res, i,i+1);
    }
    return res;
}
void extendFromCenter(string s, string& res, int left, int right) {
    while(left >=0 && right < s.size() && s[left] == s[right]) {
        left--;
        right++;
    }
    int curlen = right - left + 1 - 2;
    if(curlen > res.size()) {
        res = s.substr(left+1, curlen);
    }
}
```

53 最大子数组和 **

ans 设置为 第一个元素的值

动态规划

```
//f(i) 以第i个数字结尾的最大子数组和
f(i) = nums[i] //f(i-1)<=0
f(i) = f(i-1)+nums[i]// f(i-1)>0
```

```
int maxSubArray(vector<int>& nums) {
    int ans = nums[0], pre = nums[0];
    for(int i=1; i<nums.size(); i++) {
        pre = pre < 0 ? nums[i] : pre + nums[i];
        ans = max(ans, pre);
    }
    return ans;
}

int maxSubArray(vector<int>& nums) {
    int ans = nums[0], cursum = 0;
    for(int i=0; i<nums.size(); i++) {
        cursum += nums[i];
        ans = max(ans, cursum);
        if(cursum < 0) cursum = 0;
    }
    return ans;
}
```

54 螺旋矩阵

top bottom left right 限制移动范围

注意 第 3 4 for循环要进一步判断

```
vector<int> spiralOrder(vector<vector<int>>& matrix) {
    int left = 0, right = matrix[0].size()-1, top = 0, bottom = matrix.size()-1;
    vector<int> ans;
    while(top <= bottom && left <= right) {
        for(int i=left; i<=right; i++)
            ans.push_back(matrix[top][i]);
        for(int i=top+1; i<=bottom; i++)
            ans.push_back(matrix[i][right]);
        if(top < bottom && left < right) {
            for(int i=right-1; i>=left; i--)
                ans.push_back(matrix[bottom][i]);
            for(int i=bottom-1; i>=top+1; i--)
                ans.push_back(matrix[i][left]);
        }
```

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

或者每次都要判断 sum 防止多加入元素

主要 for循环的条件

```
vector<int> spiralOrder(vector<vector<int>>& matrix) {
    int left = 0, right = matrix[0].size()-1, top = 0, bottom = matrix.size()-1;
    int sum = (right+1) * (bottom+1);
    vector<int> ans;
    while(sum) {
        for(int i=left; i<=right && sum; i++) {
            ans.push_back(matrix[top][i]);
            sum--;
        }
        for(int i=top+1; i<=bottom && sum; i++) {
            ans.push_back(matrix[i][right]);
            sum--;
        }
        for(int i=right-1; i>=left && sum; i--) {
            ans.push_back(matrix[bottom][i]);
            sum--;
        }
        for(int i=bottom-1; i>=top+1 && sum; i--) {
            ans.push_back(matrix[i][left]);
            sum--;
        }
        left++;
        right--;
        top++;
        bottom--;
    }
    return ans;
}
```

42 接雨水

遍历两次 分别计算左边最大高度 和 右边最大高度

遍历时候从1 和 n-2开始

```
int trap(vector<int>& height) {
    vector<int> premax(height.size(),height[0]);
    vector<int> sufmax(height.size(),height[height.size()-1]);
    for(int i=1; i<height.size(); i++) {
        premax[i] = max(premax[i-1], height[i]);
    }
```

```
    for(int i=height.size()-2; i>=0; i--) {
        sufmax[i] = max(sufmax[i+1], height[i]);
    }
    int ans = 0;
    for(int i=0; i<height.size(); i++) {
        ans += min(premax[i], sufmax[i]) - height[i];
    }
    return ans;
}
```

分别记录 左边 右边的最高高度

```
int trap(vector<int>& height) {
    int premax = 0, sufmax = 0, ans = 0;
    int left = 0, right = height.size()-1;
    while(left < right) {
        premax = max(premax, height[left]);
        sufmax = max(sufmax, height[right]);
        if(premax <= sufmax) {
            ans += premax - height[left];
            left++;
        } else {
            ans += sufmax - height[right];
            right--;
        }
    }
    return ans;
}
```

11 盛最多水的容器

```
int maxArea(vector<int>& height) {
    int left = 0, right = height.size()-1;
    int ans = 0;
    while(left < right) {
        int area = (right - left) * min(height[left], height[right]);
        ans = max(ans, area);
        if(height[left] >= height[right]) right--;
        else left++;
    }
    return ans;
}
```

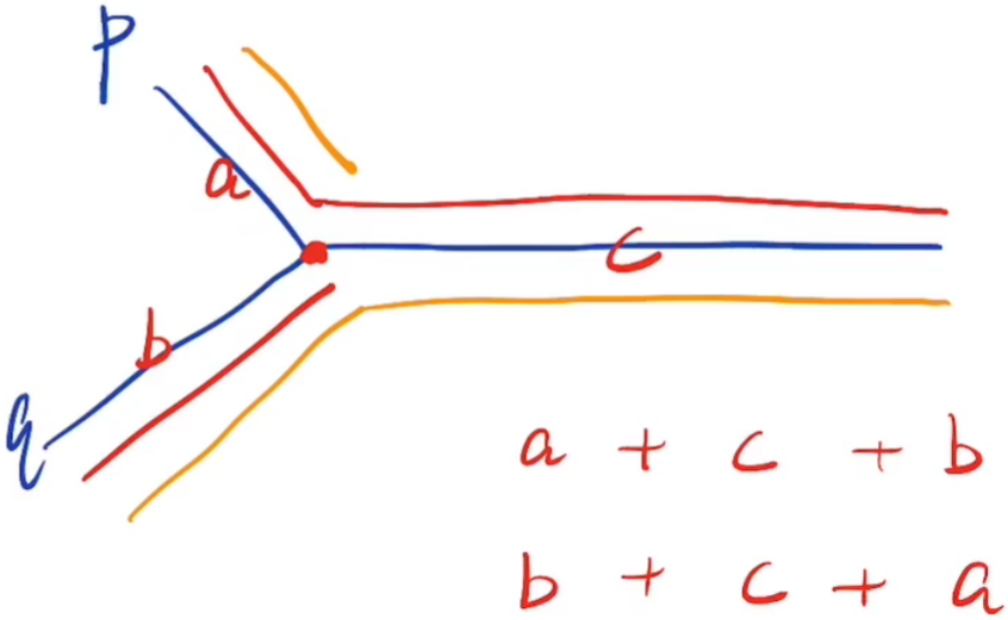
236 二叉树的最近公共祖先

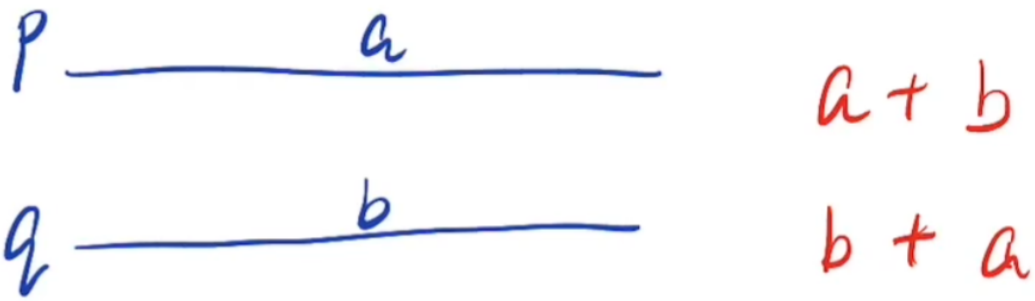
使用回溯 从下向上处理

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    if(root == NULL) return NULL;
    if(root == p || root == q) return root;
    TreeNode* left = lowestCommonAncestor(root->left, p, q);
    TreeNode* right = lowestCommonAncestor(root->right, p, q);
    if(left != NULL && right != NULL) return root;
    if(left == NULL && right != NULL) return right;
    if(left != NULL && right == NULL) return left;
    return NULL;
}
```

160 相交链表

题意 两个链表都不为空





```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    ListNode *p = headA, *q = headB;
    while(p != q) {
        p = p ? p->next : headB;
        q = q ? q->next : headA;
    }
    return q;
}
```

33 搜索旋转排序数组

判断当前位置的前半段还是后半段有序

然后再去判断target所在范围

```
int search(vector<int>& nums, int target) {
    int l=0, r=nums.size()-1;
    while(l<=r) {
        int mid = l + (r-l)/2;
        if(nums[mid] == target) return mid;
        if(nums[l] <= nums[mid]) {
            //l - mid有序
            if(nums[l]<=target && target<=nums[mid]) r=mid-1;
            else l=mid+1;
        } else {
            // mid - r 有序
            if(nums[mid]<=target && target<=nums[r]) l=mid+1;
            else r=mid-1;
        }
    }
    return -1;
}
```

200 岛屿数量

注意判断 如果true 或者是海洋 则直接返回

```
int numIslands(vector<vector<char>>& grid) {
    int m=grid.size(), n=grid[0].size();
    vector<vector<bool>> visited(m, vector<bool>(n,false));
    int ans = 0;
    for(int i=0; i<m; i++) {
        for(int j=0; j<n; j++) {
            if(visited[i][j] == false && grid[i][j] == '1') {
                ans++;
                dfs(grid, visited, i, j);
            }
        }
```

```
    }
    return ans;
}

void dfs(vector<vector<char>>& grid, vector<vector<bool>>& visited, int i, int
j) {
    int m=grid.size(), n=grid[0].size();
    if(i<0 || i>=m || j<0 || j>=n) return;
    if(visited[i][j]==true || grid[i][j]=='0') return;
    //注意这里判断 如果true 或者是海洋 则直接返回

    visited[i][j] = true;
    dfs(grid, visited, i-1, j);
    dfs(grid, visited, i+1, j);
    dfs(grid, visited, i, j-1);
    dfs(grid, visited, i, j+1);
}
```

15 三数之和

去重问题 先排序

```
vector<vector<int>> threeSum(vector<int>& nums) {
    //先排序
    sort(nums.begin(), nums.end());
    vector<vector<int>> ans;
    for(int i=0; i<nums.size(); i++) {
        if(nums[i]>0) continue;
        if(i>0 && nums[i] == nums[i-1]) continue;
        int left = i+1, right = nums.size()-1;
        while(left < right) {
            //这里没有等于 等于情况就是一个数，不符合要求 要有两个数
            int cursum = nums[i] + nums[left] + nums[right];
            if(cursum > 0) right--;
            else if(cursum < 0) left++;
            else {
                ans.push_back({nums[i], nums[left], nums[right]});
                //去重
                while(left < right && nums[right]==nums[right-1]) right--;
                while(left < right && nums[left]==nums[left+1]) left++;

                right--;
                left++;
            }
        }
    }
    return ans;
}
```

454 四数相加

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

1 两数之和

```
vector<int> twoSum(vector<int>& nums, int target) {
    unordered_map<int,int> mp;
    for(int i=0; i<nums.size(); i++) {
        auto it = mp.find(target-nums[i]);
        if(it != mp.end()) {
            return {it->second, i};
        }
        mp[nums[i]]  =i;
    }
    return { };
}
```

215 数组中第k个最大的元素

还可以引入随机

```
srand(time(0));
int i = rand() % (r - l + 1) + l;
swap(a[i], a[r]);
```

```
int findKthLargest(vector<int>& nums, int k) {
    int l=0, r=nums.size()-1, target=nums.size()-k;
    while(l < r) {
        int mid = quickselect(nums,l,r);
        if(mid == target) return nums[mid];
```

```
        else if(mid > target) r=mid-1;
        else l=mid+1;
    }
    return nums[l];
}

int quickselect(vector<int>& nums,int l, int r) {
    int i=l, j=r;
    while(i < j) {
        while(i < j && nums[j] >= nums[l]) j--;
        while(i < j && nums[i] <= nums[l]) i++;
        swap(nums[i], nums[j]);
    }
    swap(nums[i], nums[l]);
    return i;
}
```

206 反转链表

使用 pre + cur

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

92 反转链表II

如果使用上一题方法，假如left right区域很大，会遍历两次链表，所以更优方位为 **头插法**

注意

- pre从newhead开始 往后走 left-1步
- for 循环执行 right-left次
- cur一开始指向反转区间第一个节点 之后不会再修改cur，会一步一步往后移动

```
ListNode* reverseBetween(ListNode* head, int left, int right) {
    ListNode *newhead = new ListNode(0);
    newhead->next = head;
    ListNode *pre = newhead;
    for(int i=0; i<left-1; i++) {
        pre = pre->next;
    }
```

```
    ListNode *cur = pre->next, *node = nullptr;
    for(int i=0; i<right - left; i++) {
        //只用执行 right-left次
        //将node 插入到pre的后面
        node = cur->next;
        cur->next = node->next;
        node->next = pre->next;
        pre->next = node;
    }
    return newhead->next;
}
```

62 不同路径

dp

初始化二维数组

```
vector<vector<int>> dp(m, vector<int>(n,0));
```

```
int uniquePaths(int m, int n) {
    vector<vector<int>> dp(m, vector<int>(n,0));
    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];
}
```

63 不同路径 II

dp

```
int uniquePathsWithObstacles(vector<vector<int>>& obstacleGrid) {
    int m=obstacleGrid.size(), n=obstacleGrid[0].size();
    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];
}
```

343 整数拆分

dp

将 i 拆分成 j 和 i−j 的和，且i−j 不再拆分成多个正整数，此时的乘积是 j * (i−j);

将 i 拆分成 j 和 i - j 的和，且 i−j 继续拆分成多个正整数，此时的乘积是 j * dp[i−j] .

```
int integerBreak(int n) {
    vector<int> dp(n+1, 0);
    dp[2] = 1;
    for(int i=3; i<=n; i++) {
        for(int j=1; j<i; j++) {
            dp[i] = max(dp[i], max(j*(i-j), j*dp[i-j]));
        }
    }
    return dp[n];
}
```

96 不同的二叉搜索树

dp

递推公式：dp[i] += dp[j - 1] * dp[i - j];

累加过程 dp[j-1] 为 j 为头结点的左子树节点数量，dp[i-j] 为以 j 为头结点的右子树节点数量

dp[0] 需要设置为1 不然乘法都会变成 0

```
int numTrees(int n) {
    vector<int> dp(n+1,0);
    dp[0] = 1;
    for(int i=1; i<=n; i++) {
        for(int j=1; j<=i; j++) {
            dp[i] += dp[j-1] * dp[i-j];
        }
    }
    return dp[n];
}
```

509 斐波那契数

dp

```
int fib(int n) {
    if(0<=n && n <= 1) return n;
    vector<int> dp(n+1, 0);
    dp[0] = 0, dp[1] = 1;
    for(int i=2; i<=n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}
```

70 爬楼梯

dp

从i-2 走两个台阶， 从i-1走一个台阶

（如果从i-2 走两次 1 个台阶，就会和从i-1走 重复 ）

```
int climbStairs(int n) {
    if(n<=2) return n;
    vector<int> dp(n+1, 0);
    dp[1] = 1, dp[2] = 2;
    for(int i=3; i<=n; i++) {
        dp[i] = dp[i-1] + dp[i-2];
    }
    return dp[n];
}
```

升级

一步一个台阶，两个台阶，三个台阶，直到 m个台阶，有 多少种方法爬到n阶楼顶

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

746 使用最小花费爬楼梯

dp

cost[i] 是从第i个 往上爬一个 或者 两个台阶，的花费

dp[i] 到达第i个台阶的花费

根据题意理解：到达下标 0 和 1 的花费是0

```
int minCostClimbingStairs(vector<int>& cost) {
    vector<int> dp(cost.size()+1,0);
    for(int i=2; i<=cost.size(); i++) {
        dp[i] = min(dp[i-2]+cost[i-2], dp[i-1]+cost[i-1]);
    }
    return dp[cost.size()];
}
```

455 分发饼干

贪心

先排序

```
int findContentChildren(vector<int>& g, vector<int>& s) {
    sort(g.begin(), g.end());
    sort(s.begin(), s.end());
    int ans = 0;
    for(int i=0, j=0; i<s.size() && j<g.size(); i++) {
        if(s[i]>=g[j]) {
            ans++;
            j++;
        }
    }
    return ans;
}
```

007 整数反转

数学

```
int reverse(int x) {
    int ans = 0;
    while(x) {
        if(x > 0 && ans > (INT_MAX - x % 10) / 10) return 0;
        if(x < 0 && ans < (INT_MIN - x % 10) / 10) return 0;
        ans = ans * 10 + x % 10;
        x /= 10;
    }
    return ans;
}
```

003 无重复字符的最长字串

滑动窗口

删除时候用**while 并从left开始删除**

```
int lengthOfLongestSubstring(string s) {
    int len = 0, left = 0;
    unordered_set<char> st;
    for(int i=left; i<s.size(); i++) {
        while(st.count(s[i]) != 0) {
            st.erase(s[left]);
            left++;
        }
        st.insert(s[i]);
        len = max(len, i-left+1);
    }
    return len;
}
```

146 LRU缓存

要求 O(1) 则使用map + 双向链表

```
struct Node{
    int value, key;
    Node* pre;
    Node* next;
    Node():key(0),value(0),pre(nullptr),next(nullptr){}
    Node(int key_, int
value_):key(key_),value(value_),pre(nullptr),next(nullptr){}
};

class LRUCache {
private:
    unordered_map<int, Node*> mp;
    Node* head;
    Node* tail;
    int size = 0;
    int cap = 0;
public:
```

```
    LRUCache(int capacity) {
        head = new Node();
        tail = new Node();
        head->next = tail;
        tail->pre = head;
        cap = capacity;
        size = 0;
    }

    int get(int key) {
        if(mp.find(key) != mp.end()){
            Node* node = mp[key];
            RemoveFromList(node);
            AddToHead(node);
            return node->value;
        } else {
            return -1;
        }
    }

    void put(int key, int value) {
        if(mp.find(key) != mp.end()){
            Node* node = mp[key];
            RemoveFromList(node);
            AddToHead(node);
            node->value = value;
            return;
        }
        Node* node = new Node(key, value);
        mp[key] = node;
        size++;
        AddToHead(node);

        if(size > cap){
            Node* remove = RemoveLast();
            mp.erase(remove->key);
            delete remove;
            size--;
        }
    }

    void AddToHead(Node* node){
        head->next->pre = node;
        node->next = head->next;
        node->pre = head;
        head->next = node;
    }

    void RemoveFromList(Node* node){
        node->pre->next = node->next;
        node->next->pre = node->pre;
    }

    Node* RemoveLast(){
        Node* node = tail->pre;
        RemoveFromList(node);
        return node;
    }
```

```
};
```

25 K个一组翻转链表

1. ListNode* tail = pre;//循环从0计数 这里设置为前一个位置！！！

因为pre->next可能为null 即后面没有多余节点

2. ListNode* next = tail->next; //先保存后面节点 后续需要连接

3.注意pair用法 res.first res.second

```
// new
ListNode* reverseKGroup(ListNode* head, int k) {
    ListNode *newhead = new ListNode(0, head);
    ListNode *pre = newhead, *start = newhead, *end = newhead;
    while(pre->next) {
        start = start->next;//指向待反转区间的第一个
        for(int i=0; i<k; i++) {
            end = end->next;
            if(!end)
                return newhead->next;
        }
        //end移动到反转区间最后一个

        ListNode *next = end->next;
        end->next = nullptr;//保存end之后节点 并将其断开
        pair<ListNode*, ListNode*> res = myreverse(start, end);
        pre->next = res.first;//反转区间 连接回去
        pre = res.second;
        start = res.second;
        end = res.second;
        end->next = next;
    }
    return newhead->next;
}

//将 start end 区间反转，最后返回 end，start
pair<ListNode*, ListNode*> myreverse(ListNode *start, ListNode *end) {
    ListNode *newhead = new ListNode(0);
    ListNode *cur = start;
    while(cur) {
        ListNode *next = cur->next;
        cur->next = newhead->next;
        newhead->next = cur;
        cur = next;
    }
    return {end, start};
}
```

```
class Solution {
```

```
public:
    pair<ListNode*, ListNode*> myReverse(ListNode* head, ListNode* tail){
        // 翻转一个子链表，并且返回新的头与尾
        ListNode* pre = tail->next;
        ListNode* p = head;
        while(pre != tail){
            ListNode* next = p->next;
            p->next = pre;
            pre = p;
            p = next;
        }
        return {tail, head};
    }
    ListNode* reverseKGroup(ListNode* head, int k) {
        ListNode* hair = new ListNode(0);
        hair->next = head;
        ListNode* pre = hair;

        while(head) {
            ListNode* tail = pre;//循环从0计数 这里设置为前一个位置 ！！！
            // 而且 tail->next 可能为null ！！！

            // 看剩下部分长度是否大于等于k
            for(int i=0; i<k; i++){
                tail = tail->next;
                if(tail == nullptr) {
                    return hair->next;
                }
            }
            ListNode* next = tail->next; //先保存后面节点 后续需要连接
            pair<ListNode*,ListNode*> result = myReverse(head, tail);
            head = result.first; // first  second 注意用法
            tail = result.second;
            //子链表重新接回原链表
            pre->next = head;
            tail->next = next;
            pre = tail;
            head = tail->next;
        }
        return hair->next;
    }
};
```