1、快慢指针找环

为什么快指针和慢指针总是会在环中某一点相遇，而不会刚好“跳过”彼此?
由于快指针每次跳两步，慢指针每次跳一步，快指针实际上比慢指针多走了一步（两步-一步）。每当快指针和慢指针没有相遇时，快指针相当于以慢指针的速度“追”慢指针。

```
if(!head || !head->next) return false;
ListNode *fast = head, *low = head;
while (fast && fast->next) {
    fast = fast->next;
    fast = fast->next;
    low = low->next;
    if(fast == low ) return true;
}
return false;
```

2、两数之和

1. 排序、左右指针、向中间移动。根据大小关系左指右移或者右指左移。
2. hash.

3、两数相加（链表）

listNode val carry.

4、无重复字符的最长子串

贪心、hash、左右指针。右指针往右走，一旦遇到重复字符，左指针右移动hash值减一，判断最大值。

5、两个正序数组的中位数

```
if (left < m && (right >= n || nums1[left] <= nums2[right])) {
    curr = nums1[left++];
} else {
    curr = nums2[right++];
}
```

6、最长回文子串

1. 暴力（n*3）不行

2. 动态规划
   第一层循环用len，第二层循环用左端点

7、正则表达式匹配

```
// 初始化f[0][j]，处理模式以 '*' 开头的情况
for (int j = 2; j <= n; j++) {
    if (p[j] == '*' && f[0][j - 2]) {
        f[0][j] = true;  // '*' 匹配空字符串
    }
}

//处理f[i][j]
if (p[j] == s[i] || p[j] == '.') {
    f[i][j] = f[i - 1][j - 1];  // 匹配单个字符或 '.'
} else if (p[j] == '*') {
    // '*' 匹配 0 或多个字符
    f[i][j] = f[i][j - 2] ||  // '*' 匹配 0 次
        f[i - 1][j - 2] && (p[j - 1] == s[i] || p[j - 1] == '.') ||
        f[i - 1][j] && (p[j - 1] == s[i] || p[j - 1] == '.');
        // 前面两行可以缩写为：(f[i - 1][j] && (p[j - 1] == s[i] || p[j - 1] == '.'));  // '*' 匹
}
```

8、盛最多水的容器

双指针向中间靠拢，谁低谁移动

9、三数之和

1. 排序
2. 一次循环+双指针（在循环右侧）
3. 符合条件时。若下一个是重复，一直移动到最后一个重复的。两个指针都动一下

10、电话号码的数字组合

```
void dfs(const string& digits, int cur ,string word, vector<string>& ret,const vector<string>& l
    if (cur == digits.size()) ret.push_back(word);
    else {
        int index = digits[cur] - '0';
        for (int i = 0; i < list[index].size(); i++) {
            dfs(digits, cur + 1,word + list[index][i], ret, list);
        }
    }
}

dfs(digits, 0 ,"", ret, list);
```

11、删除倒数第n个节点

1. 快慢指针间隔n
2. 计算长度减去n

12、合并两个有序链表

建新节点取小节点

13、括号生成

1. 终止：L == 0 && R == 0
2. 任何时候：L <= R

```
//风格一
string buf;
void dfs(int left, int right) {
    if (left == 0 && right == 0) {
        ret.push_back(buf);return;
    }
    if (left > 0) {
        buf.push_back('(');dfs(left - 1, right);buf.pop_back();
    }
    if (left < right) {
        buf.push_back(')');dfs(left, right - 1);buf.pop_back();
    }
}

//风格二
char buf[20];
void dfs(int left, int right, int cur) {
    if ( !left && !right ) {
        ret.push_back(buf);return;
    }
    if (left ) {
        buf[cur] = '(';dfs(left - 1, right, cur + 1);
    }
    if (left < right) {
        buf[cur] = ')';dfs(left, right - 1,cur + 1 );
    }
}
```

14、合并K个升序链表

- 优先队列取链表数组的每一个链表的头
- while (!pq.empty())
- 拿走top之后，放该节点的下一个节点

```
typedef pair<int, ListNode*> Elem;
using Elem = pair<int, ListNode*>;

priority_queue<Elem, vector<Elem>, greater<Elem>> pq;
//greater是小根堆，取出来是最小值 less...
```

15、下一个排列

1. 从右到左，找到第一个a[i] < a[j]，若没有就直接反转整个vector
2. 从右往左找到第一个a[j] > a[i]（至少有一个a[j+1]）
3. 交换a[i]和a[j]，然后反转i + 1到最后一个元素

16、最长有效括号

1. 栈里面存括号的index，提前存一个-1
2. 遇到“（”就入栈。遇到“）”判断栈顶是不是-1或者栈顶是“）”也入栈，否则就pop，然后max(ret, i-stk.top())

17、在排序数组中查找元素的第一个和最后一个位置

```
int bin1(const vector<int>& nums, int &target) {
    int left = 0, right = nums.size() - 1;
    int ans = -1;
    while (left <= right) {
        int mid = (left + right ) >> 1;
        if (nums[mid] < target) left = mid + 1;
        else  {
            ans = mid;
            right = mid - 1;
        }
    }
    if (ans == -1 || nums[ans] != target) return -1;
    return ans;
}
int bin2(const vector<int>& nums, int &target) {
    int left = 0, right = nums.size() - 1;
    int ans = -1;
    while (left <= right) {
        int mid = (left + right ) >> 1;
        if (nums[mid] > target) right = mid - 1;
        else  {
            ans = mid;
            left = mid + 1;
        }
    }
    if (ans == -1 || nums[ans] != target) return -1;
    return ans;
}
```

17、只出现一次的数字

位运算 ^ (异或) 相同为0，不同为1

```
0^a^a^b^b^...^x
=0^0^...^x
=x
```

满足交换律，顺序无所谓

18、搜索插入的位置

19、回文链表

1. 复制出来新的链表（倒置）
2. 快慢指针，慢指针到中间节点，转置后半链表

20、倒置链表

```
while(head) {
    newHead = head;
    newHead->next = p;
    p = newHead;
    head = head->next;  //注意此项的位置，不能在第二行的上面，否则就是超时（循环）
}
```

21、轮转数组

- 和交换前几个元素和后几个元素的位置的题的区别在于要取模
- reverse(a, b) 左闭右开

22、合并区间

1. 按照左端点排序
2. 更新L和R

23、寻找重复数

nums中的数作为index，使index位置的数字发生变化（变负数或者大于n的数字）

24、颜色分类

1. 循环两遍
2. 双指针

```
//注意看对于指针指向2的情况，还有终止条件
while (i <= right) {
    if (nums[i] == 0) {
        swap(nums[i], nums[left]);
        i++;left++;
    }else if (nums[i] == 2){
        swap(nums[i], nums[right]);right--;
    }
    else i++;
}
```

25、最小栈

1. 从零开始，可以注意实现扩展数组
2. 用两个栈实现，一个正常，另外一个有可能装当前数字（栈顶数字一定是最小的），如果等于栈顶数字也装进去

26、最大子序和

- 如果sum > 0，则说明 sum 对结果有增益效果，则 sum 保留并加上当前遍历数字
- 如果sum <= 0，则说明 sum 对结果无增益效果，需要舍弃，则 sum 直接更新为当前遍历数字
- 每次比较sum和ans的大小，将最大值置为ans，遍历结束返回结果

27、树的直径

其实就是调用求树的深度代码，在里面加一个ret = max(ret, left + right)

28、对称二叉树

很多时候一个递归函数是不用的，写一个额外的递归函数，再在主函数里面调用就可以了

```
bool check(TreeNode* left, TreeNode* right) {
    if (!left && !right) return true;
    if (!left || !right) return false;
    return left->val == right->val && check(left->left, right->right) && check(left->right, righ
}
bool isSymmetric(TreeNode* root) {
    return check(root->left, root->right);
}
```

29、将有序数组转换为二叉搜索树

- 中序搜索
- 取数组index的中间值作为根节点
- 一个主函数调用一个递归函数

30、验证二叉搜索树

- 利用的是中序遍历的所有值是递增的

```
long long INF_MIN = LONG_MIN;
bool isValidBST(TreeNode* root) {
    if (!root) return true;
    bool l = isValidBST(root->left);
    if (root->val <= INF_MIN) return false;
    INF_MIN = root->val;
    return l && isValidBST(root->right);
}
```

31、BST中第k小的元素

和30类似，用中序遍历，更新count

32、组合总和

```
class Solution {
public:
    vector<vector<int>> ret;
    vector<int> b;

    void dfs(vector<int> &candidates, int p, int sum, int target) {
        if (p == candidates.size()) {
            return;
        }
        if (sum == target) {
            ret.push_back(b);
            return;
        }
        dfs(candidates, p + 1, sum, target);

        if (sum + candidates[p] <= target) {
            b.push_back(candidates[p]);
            dfs(candidates, p, sum + candidates[p], target);
            b.pop_back();
        }

    }
    vector<vector<int>> combinationSum(vector<int>& candidates, int target) {
        dfs(candidates, 0, 0, target);
        return ret;
    }
};
```

33、字母异位词分组

- 注意重复的也要
- map<string vector>
- 排完序当key,原值放value中

34、跳跃游戏

1. 左指针记录当前位置，右指针是最远能到的位置，两指针相遇代表不行
2. dp: bool f[i] 表示从这个位置能否到末尾，从右往左遍历

35、不同路径

dp:f[i][j]表示f[0]f[0]到当前位置的路径数量 **(f[1][1] = 1)**

36、最小路径和

与35类似,区别在于如何更新（之前都是加1，现在是加位置上面的数字）

37、编辑距离

```
if (word1[i-1] == word2[j-1]) f[i][j] = f[i-1][j-1];
else f[i][j] = f[i-1][j-1] + 1;
f[i][j] = min(f[i][j],f[i-1][j] +1);
f[i][j] = min(f[i][j], f[i][j-1] +1);
```

38、二叉树展开为链表(不太能理解为什么要备份)

```
TreeNode* pre;
void dfs(TreeNode* root) {
    if(!root) return;

    pre->right = root;
    pre->left = nullptr;

    TreeNode* left = root->left;
    TreeNode* right = root->right;

    pre = root;
    dfs(left);
    dfs(right);
}
```

39、二叉树的层序遍历

qu不为空就遍历，获取当前size，for遍历输入当前值到vector、输入孩子到qu

40、岛屿数量

深度搜索，走过的改值，扫描上下左右

41、实现前缀树（Trie）

26叉树

```
class Trie {
private:
    bool isEnd;
    Trie* next[26];
public:
    Trie() {
        this->isEnd = false;
        for (int i= 0;i < 26; i++) {
            this->next[i] = nullptr;
        }
    }

    void insert(string word) {
        Trie* node = this;
        for (char c : word) {
            if (node->next[c - 'a'] == nullptr) node->next[c - 'a'] = new Trie();
            node = node->next[c - 'a'];
        }
        node->isEnd = true;
    }

    bool search(string word) {
        Trie* node = this;
        for (char c : word) {
            node = node->next[c - 'a'];
            if (node == nullptr) {
                return false;
            }
        }
        return node->isEnd;
    }
```

42、输出链表中倒数k的节点及其链表

```
if (!pHead) return nullptr;
ListNode *fast = pHead, *slow = pHead;
while (k--) {
    if (fast == nullptr) return nullptr;
    fast = fast->next;
}

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

43、合并二叉树

```
TreeNode* mergeTrees(TreeNode* t1, TreeNode* t2) {
    // write code here
    if (!t1) return t2;
    if (!t2 ) return t1;
    t1->val += t2->val;
    t1->left = mergeTrees(t1->left, t2->left);
    t1->right = mergeTrees(t1->right, t2->right);
    return t1;
}
```

44、字符串变形

- 将字符串大小写反转
- 将整个字符串的所有单词位置反转
  思路：

1. stack
2. 先都反转，然后按照遇到空格反转单词 （空间复杂度常数级）

45、最长公共前缀

- 注意外层的for循环是第一个字符串的长度

46、最长连续序列

- 用set
- string.substr(a,len)，第二个参数是字串长度（此题可不用这个函数）

47、找到字符串中所有字母异位词

- 滑动窗口
- 构造26长度的vector 这个容器重载了==运算符，可以直接用

48、缺失的第一个正数

1. 数组长度为N，那么缺失的第一个数字一定在[1, N+1]中。
   并且所有的数字是1 ~ N的时候，第一个缺失的正数才是N + 1
2. 遍历数组，所有小于等于0的数字变为 N + 1
3. 遍历数组，对于数字a，若abs(a)小于等于N，将nums[a-1] = -abs(nums[a-1])
4. 遍历数组，对于第一个出现的正数，返回其下标+1，若没有就返回N+1

49、除自身以外数组的乘积

方法

1. 两个辅助数组一个结果数组，辅助数组各自维护该位置数组元素的左边乘积和右边乘积
2. 只有一个结果数组，第一遍循环保存该位置对应数字的左边乘积，第二遍循环借助辅助变量R = 1，辅助数组元素 * R，然后用R保存当前位置右边乘积

50、和为 K 的子数组

1. 暴力循环
2. 哈希表

```
class Solution {
public:
    int subarraySum(vector<int>& nums, int k) {
        unordered_map<int, int> mp;
        mp[0] = 1;
        int count = 0, pre = 0;
        for (auto& x:nums) {
            pre += x;
            if (mp.find(pre - k) != mp.end()) {
                count += mp[pre - k];
            }
            mp[pre]++;
        }
        return count;
    }
};
```

51、矩阵置0

1. 借助标记数组或者set存标志行列：T(n) = O(mn),S(n) = O(m + n)
2. 直接将标志位放到第一行第一列，S(n) = O(1)

52、螺旋矩阵

1. 方向数组、方向指示变量、访问标志
2. 下一个数字不合法，更新方向指示变量，每访问一个数字，修正为已访问

53、单词搜索

别忘了标志已经访问，然后再撤回就行

54、旋转图像

方法：

1. 辅助二维数组
2. 先水平翻转，再主对角线翻转

55、搜索二维矩阵 II

方法：

1. 每一行都用折半查找
2. Z字查找（从右上角开始判断）

56、每日温度

1. 两层for循环（超时）
2. 建立一个stack存下标 !s.empty() && temperatures[i] > temperatures[s.top()]出栈，否则就存i

56、数组中的第K个最大元素

- 线性时间选择

```
int partition(vector<int>& A, int low, int high) {
    int pivot = A[low];
    while (low < high) {
        while (low < high && A[high] >= pivot) {
            high--;
        }
        A[low] = A[high];
        while (low < high && A[low] <= pivot) {
            low++;
        }
        A[high] = A[low];
    }
    A[low] = pivot;
    return low;
}
```

57、前 K 个高频元素

方法:

1. 用map + priority_queue(堆)
2. 用map + vector<pair<int, int>>存，然后排序

58、两两交换链表中的节点

一个pre一个p，一个temp一个tail

59、K 个一组翻转链表

1. 写反转整个链表的函数，提供前一个节点和下一个节点和链表本身
2. 计数，一到k就用反转函数

59、搜索二维矩阵

1. 采用Z字查找
2. 一次折半查找（用n映射i和j）

60、旋转数组中的最小数

- 用mid和high的关系做判定
- 最小值的

61、打家劫舍

- dp:不要忘了 `ret[1] = max(nums[1], nums[0])` 而不是 `ret[1] = nums[1]`

62、完全平方数

方法：

1. dp

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

2. 四平方和定理：当且仅当 n ≠ 4^k*(8m+7) 时，n 可以被表示为至多三个正整数的平方和。因此，当 4^k*(8m+7) 时，n 只能被表示为四个正整数的平方和。前一种情况：为1时，n是完全平方数，为2时枚举a，判断n-a^2是不是完全平方数，排除求3。

63、零钱兑换

```
int len  = coins.size();
vector<int> ret(amount+1, amount +1);
ret[0] = 0;
for (int i=1; i <= amount; i++) {
    for (int j = 0; j < len; j++) {
        if (coins[j] <= i) {
            ret[i] =min(ret[i], ret[i - coins[j]] + 1);
        }
    }
}
return ret[amount] > amount ? -1 : ret[amount];
```

64、单词拆分

```
unordered_set<string> set; //vector没有find函数，并且整个过滤重复单词
for (string str : wordDict) {
    set.insert(str);
}
vector<bool> dp(s.size() + 1, false);
dp[0] = true;
for (int i = 1; i <= s.size(); i++) {
    for (int j = 0; j < i; j++) {
        if (dp[j] && set.find(s.substr(j, i - j)) != set.end()) {
            dp[i] = true;
            break;
        }
    }
}
```

65、最长递增子序列

1. dp：

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

2. 贪心+二分：

如果 `nums[i]>d[len]` ，则直接加入到 d 数组末尾，并更新 `len=len+1` ；

否则，在 d 数组中二分查找，找到第一个比 `nums[i]` 小的数 `d[k]` ，并更新 `d[k+1]=nums[i]` 。

```
int len = 1, n = (int)nums.size();
if (n == 0) {
    return 0;
}
vector<int> d(n + 1, 0);
d[len] = nums[0];
for (int i = 1; i < n; ++i) {
    if (nums[i] > d[len]) {
        d[++len] = nums[i];
    } else {
        int l = 1, r = len, pos = 0; // 如果找不到说明所有的数都比 nums[i] 大，此时要更新 d[1]，所
        while (l <= r) {
            int mid = (l + r) >> 1;
            if (d[mid] < nums[i]) {
                pos = mid;
                l = mid + 1;
            } else {
                r = mid - 1;
            }
        }
        d[pos + 1] = nums[i];
    }
}
return len;
```

66、乘积最大子数组

两种dp

1. 一个minF,一个maxF

```
long maxF = nums[0], minF = nums[0], ans = nums[0];
for (int i = 1; i < nums.size(); ++i) {
    long mx = maxF, mn = minF;
    maxF = max(mx * nums[i], max((long)nums[i], mn * nums[i]));
    minF = min(mn * nums[i], min((long)nums[i], mx * nums[i]));
    ans = max(maxF, ans);
}
return ans;
```

2. 从左往右，从右往左： `example:[2,3,-2,4]`

```
int product = 1, n = nums.length;
int max = nums[0];

for(int i = 0;i < n;i++){
    product *= nums[i];
    max = Math.max(max, product);
    if(nums[i] == 0){
        product = 1;
    }
}
product = 1;
for(int i = n - 1;i >= 0;i--){
    product *= nums[i];
    max = Math.max(max, product);
    if(nums[i] == 0){
        product = 1;
    }
}
return max;
```

67、全排列

1. 确定一个然后标记，枚举未访问的下一个

```
void dfs(const vector<int>& nums, int cur) {
    if (cur >= nums.size()) {
        ret.push_back(p);
        return;
    }
    for (int i = 0; i <nums.size(); i++) {
        if (!visited[i]) {
            visited[i] = true;
            p[cur] = nums[i];
            dfs(nums, cur + 1);
            visited[i] = false;
        }
    }
}
```

2. swap(nums[cur], i)

```
void dfs(vector<int>& nums, int cur) {
    if (cur >= nums.size()) {
        ret.push_back(nums);
        return;
    }
    for (int i = cur; i <nums.size(); i++) {
        swap(nums[i], nums[cur]);
        dfs(nums, cur + 1);
        swap(nums[i], nums[cur]);
    }
}
```

3、下一个排列

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

do {
    ret.push_back(nums);

}while(next_permutation(nums.begin(), nums.end()));
return ret;
```

68、子集

1. 递归，当前元素选或者不选

```
void dfs(vector<int>& nums, int cur) {
    if (cur >= nums.size()) {
        ret.push_back(p);
        return;
    }
    dfs(nums, cur +1);
    p.push_back(nums[cur]);
    dfs(nums, cur + 1);
    p.pop_back();
}
```

69、分割回文串

f[i][j]: i到j是不是回文串
循环中是dfs(s, i + 1)，不是dfs(s, cur + 1)

```
vector<vector<bool>> f;
vector<vector<string>> ret;
vector<string> ans;
int n;

void dfs(string s, int cur) {
    if (cur >= s.size()) {
        ret.push_back(ans);
        return;
    }
    for (int i = cur; i < s.size(); i++) {
        if (f[cur][i]) {
            ans.push_back(s.substr(cur, i - cur + 1));
            dfs(s, i + 1);
            ans.pop_back();
        }
    }
}
vector<vector<string>> partition(string s) {
    n = s.size();
    f.assign(n, vector<bool>(n ,true));
    for (int i = n-1; i >= 0; i--) {
        for (int j = i + 1; j < n; j++) {
            f[i][j] = (f[i + 1][j - 1]) && s[i]== s[j];
        }
    }
    dfs(s, 0);
    return ret;
}
```

70、二叉树的右视图

层序遍历，queue中的最后一个元素

71、从前序与中序遍历序列构造二叉树

1. `l1 <= r1` 就执行 `new TreeNode(preorder[l1])`
2. 用 `auto it = find(inorder.begin(), inorder.end(), preorder[l1])` 和 `int index = int(it-inorder.begin())` 定位

72、路径总和 III

1. 深搜（两次递归，一次是求当前节点往下走有没有，另外一个是整个树的递归）

```
int rootSum(TreeNode* root, int targetSum) {
    if (!root) {
        return 0;
    }

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

    ret += rootSum(root->left, targetSum - root->val);
    ret += rootSum(root->right, targetSum - root->val);
    return ret;
}

int pathSum(TreeNode* root, int targetSum) {
    if (!root) {
        return 0;
    }

    int ret = rootSum(root, targetSum);
    ret += pathSum(root->left, targetSum);
    ret += pathSum(root->right, targetSum);
    return ret;
}
```

2. 前缀和

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

    int ret = 0;
    curr += root->val;
    if (prefix.find(curr - targetSum)!= prefix.end()) {
        ret = prefix[curr - targetSum];
    }

    prefix[curr]++;
    ret += dfs(root->left, curr, targetSum);
    ret += dfs(root->right, curr, targetSum);
    prefix[curr]--;

    return ret;
}
```

73、二叉树的最近公共祖先

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

74、二叉树中的最大路径和

1. 判断左右子树对应值和0的大小关系
2. 判断当前值+左右子树值
3. 返回当前值+max(左右子树值)

75、最长公共子序列

```
if (text1[i-1] == text2[j-1]) {
    f[i][j] = f[i-1][j-1] + 1;
}else {
    f[i][j] = max(f[i-1][j], f[i][j-1]);
}
```

76、接雨水

max_left记录l往右边走的时走的最高的柱子，max_left-l就是答案

```
int max_left = 0;
int max_right = 0;
int l = 0, r = height.size()-1;
while (l < r) {
    max_left = max(max_left, height[l]);
    max_right = max(max_right, height[r]);
    if (height[l] < height[r]) {
        ans += max_left - height[l];
        l++;
    }else {
        ans +=max_right - height[r];
        r--;
    }
}
return ans;
```

77、滑动窗口最大值

1. 优先队列pq<pair<int, int>>（nums[i],i）：先将前k个值放进去，往下再走的时候，先push，然后对于top()元素检查，如果top().second <= i-k，出队列，然后拿当前top()元素入ret
2. 只用一个队列存index，如果 `!q.empty() && nums[i] >= nums[q.back()]` 一直 pop，这样保证q.front一直是最大的。 `while (qu.front() <= i - k) {qu.pop_front();}` 在这个保证当前的front是有效的

78、复制带随机指针的链表

1. hash记录当前节点是否存在，不存在就创建新的并记录

```
Node* copyRandomList(Node* head) {
    // if (!head) return nullptr;
    // if (hashmap.find(head) == hashmap.end()) {
    //     Node* newHead = new Node(head->val);
    //     hashmap[head] = newHead;
    //     newHead->next = copyRandomList(head->next);
    //     newHead->random = copyRandomList(head->random);
    // }
    // return hashmap[head];
    //或者
    unordered_map<Node*,Node*>hmap;
    Node* copyRandomList(Node* head) {
        Node *p=head;
        while(p){
                        hmap.insert({p,new Node(p->val)});
            p=p->next;
                }
                p=head;
                while(p){
                        hmap[p]->next=hmap[p->next];
                        hmap[p]->random=hmap[p->random];
            p=p->next;
                }
                return hmap[head];
    }
}
```

2.

- 拷贝前一个结点，n-->2n
- 设置random节点
- 分离链表

```cpp
class Solution {
public:
    Node* copyRandomList(Node* head) {
        if (head == nullptr) {
            return nullptr;
        }
        for (Node* node = head; node != nullptr; node = node->next->next) {
            Node* nodeNew = new Node(node->val);
            nodeNew->next = node->next;
            node->next = nodeNew;
        }
        for (Node* node = head; node != nullptr; node = node->next->next) {
            Node* nodeNew = node->next;
            nodeNew->random = (node->random != nullptr) ? node->random->next : nullptr;
        }
        Node* headNew = head->next;
        for (Node* node = head; node != nullptr; node = node->next) {
            Node* nodeNew = node->next;
            node->next = node->next->next;
            nodeNew->next = (nodeNew->next != nullptr) ? nodeNew->next->next : nullptr;
        }
        return headNew;
    }
};
```

79、课程表

邻接表+拓扑排序

```
bool canFinish(int numCourses, vector<vector<int>>& pre) {
    vector<vector<int>> edges;
    vector<int> indeg;
    edges.resize(numCourses);
    indeg.resize(numCourses);
    for (int i = 0; i <pre.size(); i++ ) {
        edges[pre[i][1]].push_back(pre[i][0]);
        indeg[pre[i][0]]++;
    }
    queue<int> qu;
    for (int i = 0; i < numCourses;i++) {
        if (indeg[i] == 0) qu.push(i);
    }
    int count =0 ;
    while(!qu.empty()) {
        count++;
        int p = qu.front();
        qu.pop();
        for (int i : edges[p]) {
            indeg[i]--;
            if (indeg[i] == 0) qu.push(i);
        }
    }

    return count == numCourses;

}
```

80、烂橘子

- 多源广搜
- 烂橘子入队、统计新鲜橘子
- `freshCount>0 && !q.empty()` 就遍历四个方向，更新time、count，有新的烂橘子就入队
- `freshCount` 符不符合要求

```cpp
int orangesRotting(vector<vector<int>>& grid) {
    int m = grid.size();        // 行数
    int n = grid[0].size();     // 列数
    queue<pair<int, int>> q;    // BFS 队列
    int freshCount = 0;         // 统计新鲜橘子数量

    // 遍历网格，记录初始腐烂橘子的位置，并统计新鲜橘子数量
    for (int i = 0; i < m; i++) {
        for (int j = 0; j < n; j++) {
            if (grid[i][j] == 2) {
                q.push({i, j}); // 将腐烂橘子入队
            } else if (grid[i][j] == 1) {
                freshCount++;   // 统计新鲜橘子数量
            }
        }
    }

    // 如果没有新鲜橘子，直接返回 0
    if (freshCount == 0) return 0;

    int time = 0;               // 记录腐烂所需时间
    vector<vector<int>> dir = {{0, 1}, {0, -1}, {1, 0}, {-1, 0}}; // 方向数组

    // BFS 遍历
    while (freshCount>0 && !q.empty()) {
        int size = q.size();
        // 如果有橘子腐烂，时间加 1
        time++;

        for (int i = 0; i < size; i++) {
            auto [x, y] = q.front();
            q.pop();

            // 遍历四个方向
            for (const auto& d : dir) {
                int xNext = x + d[0];
                int yNext = y + d[1];

                // 检查边界和是否为新鲜橘子
                if (xNext >= 0 && yNext >= 0 && xNext < m && yNext < n && grid[xNext][yNext] ==
                    grid[xNext][yNext] = 2;  // 腐烂新鲜橘子
                    q.push({xNext, yNext});  // 将新的腐烂橘子加入队列
                    freshCount--;            // 新鲜橘子数量减少
```

```
                }
            }
        }


    }

    // 如果还有新鲜橘子，返回 -1，否则返回腐烂所需的时间
    return freshCount == 0 ? time : -1;
}
```

81、八皇后

```
class Solution {
public:
    vector<vector<string>> ret;
    bool isValid(const vector<string>& chessboard, int row, int column, int n) {
        for (int i = 0; i < n;i++) {
            if(chessboard[row][i] == 'Q' || chessboard[i][column] == 'Q') return false;
        }
        // 检查左上主对角线 (\ 方向)
        for (int i = row - 1, j = column - 1; i >= 0 && j >= 0; i--, j--) {
            if (chessboard[i][j] == 'Q') return false;
        }

        // 检查右上副对角线 (/ 方向)
        for (int i = row - 1, j = column + 1; i >= 0 && j < n; i--, j++) {
            if (chessboard[i][j] == 'Q') return false;
        }
        return true;

    }
    void dfs(vector<string>& chessboard, int n, int row) {
        if (row==n) {
            ret.push_back(chessboard);
            return;
        }
        for (int i = 0; i < n;i++) {
            if(isValid(chessboard, row,i,n)) {
                chessboard[row][i] = 'Q';
                dfs(chessboard, n, row+1);
                chessboard[row][i] = '.';
            }
        }
    }
    vector<vector<string>> solveNQueens(int n) {
        vector<string> chessboard(n, string(n,'.'));
        dfs(chessboard, n, 0);
        return ret;
    }
};
```

82、划分字母区间

```
vector<int> partitionLabels(string s) {
    int last[26];
    int length = s.size();
    for (int i = 0; i < length; i++) {
        last[s[i] - 'a'] = i;
    }
    vector<int> partition;
    int start = 0, end = 0;
    for (int i = 0; i < length; i++) {
        end = max(end, last[s[i] - 'a']);
        if (i == end) {
            partition.push_back(end - start + 1);
            start = end + 1;
        }
    }
    return partition;
}
```

83、最小覆盖子串

```
unordered_map<char, int> need, window;

string minWindow(string s, string t) {
    for(char c : t) {
        need[c]++;
    }
    string ans;
    for (int i = 0,j = 0,count = 0; i < s.size(); i++) {
        if(++window[s[i]] <= need[s[i]]) count++;
        while (window[s[j]] > need[s[j]]) window[s[j++]]--;
        if (count == t.size()) {
            if (ans.empty() || ans.size() > i - j + 1) {
                ans = s.substr(j, i-j+1);
            }
        }
    }
    return ans;
}
```

84、合并链表

```
class Solution {
public:
    ListNode* merge(ListNode* head1, ListNode* head2) {
        ListNode* newHead = new ListNode(0);
        ListNode* p1 = head1, *p2 = head2, *p = newHead;
        while (p1 && p2) {
            if (p1->val < p2->val) {
                p->next = p1;
                p1 = p1->next;
            }else {
                p->next = p2;
                p2 = p2->next;
            }
            p = p->next;
        }
        if(p1) p->next = p1;
        if(p2) p->next = p2;
        return newHead->next;
    }
    ListNode* sortList(ListNode* head) {
        if (head == nullptr) return head;
        int len = 0;
        for (ListNode* p = head; p;p = p->next) len++;

        ListNode* newHead = new ListNode(0);
        newHead->next = head;

        for (int i = 1;i < len;i *= 2) {
            ListNode* pre = newHead, *cur = newHead->next;
            while (cur) {
                ListNode *head1 = cur;
                for (int j = 1; j < i && cur->next; j++) {
                    cur = cur->next;
                }

                ListNode * head2 = cur->next;
                cur->next = nullptr;
                cur = head2;

                for (int j = 1; j < i && cur && cur->next; j++) {
                    cur = cur->next;
```

```
                }
                ListNode *next = nullptr;
                if (cur) {
                    next = cur->next;
                    cur->next = nullptr;
                }

                ListNode* merged = merge(head1, head2);
                pre->next = merged;
                while (pre->next) {
                    pre = pre->next;
                }
                cur = next;
            }
        }
        return newHead->next;
    }
};
```

85、跳跃游戏II

```
int jump(vector<int>& nums) {
    if(nums.size() == 1) return 0;
    int cur = 0, next = 0, ret = 0;
    for (int i = 0; i < nums.size() - 1; i++) {
        next = max(next, i+ nums[i]);
        if (i == cur) {
            ret++;
            cur = next;
        }
    }
    return ret;
}
```

86、单词拆分

```
bool wordBreak(string s, vector<string>& wordDict) {
    int len = s.size();
    vector<bool> dp(len + 1, false);
    dp[0] = true;

    for (int i = 1; i <= len;i++) {
        for (string str : wordDict) {
            int str_size = str.size();
            if (i + str_size-1 <= len && s.substr(i-1, str_size) == str && dp[i-1]) {
                dp[i-1 + str_size] = true;
            }
        }
    }

    return dp[len];
}
```

87、数据流的中位数

```
void addNum(int num) {
    if (queMin.empty() || num <= queMin.top()) {
        queMin.push(num);
        if (queMax.size() + 1 < queMin.size()) {
            queMax.push(queMin.top());
            queMin.pop();
        }
    } else {
        queMax.push(num);
        if (queMax.size() - 1 > queMin.size()) {
            queMin.push(queMax.top());
            queMax.pop();
        }
    }
}

double findMedian() {
    if (queMax.size() > queMin.size()) return queMax.top();
    else if (queMax.size() < queMin.size()) return queMin.top();
    else return (queMax.top() + queMin.top()) / 2.0;
}
```

88、分割等和子集

1. 二维数组

```cpp
class Solution {
public:
    bool canPartition(vector<int>& nums) {
        int n = nums.size();
        if (n < 2) {
            return false;
        }
        int sum = accumulate(nums.begin(), nums.end(), 0);
        int maxNum = *max_element(nums.begin(), nums.end());
        if (sum & 1) {
            return false;
        }
        int target = sum / 2;
        if (maxNum > target) {
            return false;
        }
        vector<vector<int>> dp(n, vector<int>(target + 1, 0));
        for (int i = 0; i < n; i++) {
            dp[i][0] = true;
        }
        dp[0][nums[0]] = true;
        for (int i = 1; i < n; i++) {
            int num = nums[i];
            for (int j = 1; j <= target; j++) {
                if (j >= num) {
                    dp[i][j] = dp[i - 1][j] | dp[i - 1][j - num];
                } else {
                    dp[i][j] = dp[i - 1][j];
                }
            }
        }
        return dp[n - 1][target];
    }
};
```

2.

```cpp
class Solution {
public:
    bool canPartition(vector<int>& nums) {
        int n = nums.size();
        if (n < 2) {
            return false;
        }
        int sum = 0, maxNum = 0;
        for (auto& num : nums) {
            sum += num;
            maxNum = max(maxNum, num);
        }
        if (sum & 1) {
            return false;
        }
        int target = sum / 2;
        if (maxNum > target) {
            return false;
        }
        vector<int> dp(target + 1, 0);
        dp[0] = true;
        for (int i = 0; i < n; i++) {
            int num = nums[i];
            for (int j = target; j >= num; --j) {
                dp[j] |= dp[j - num];
            }
        }
        return dp[target];
    }
};
```