

# LeetCode

— SetoWen

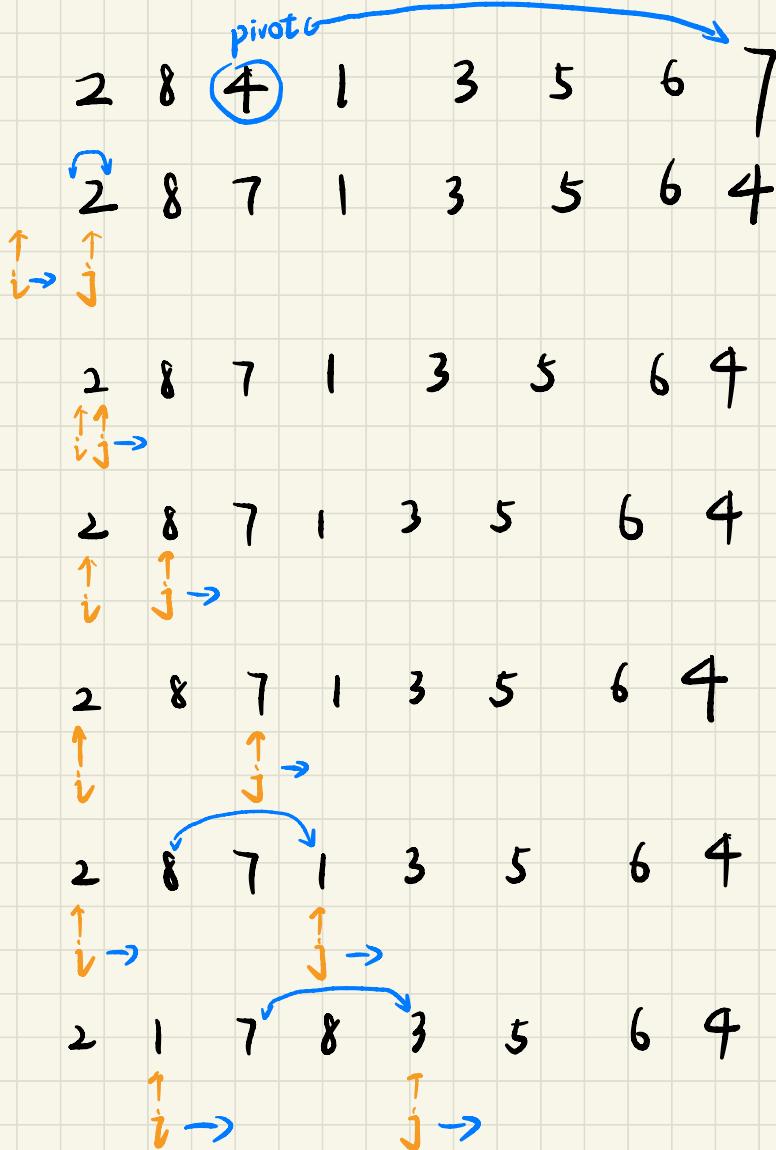
# 1. 排序算法

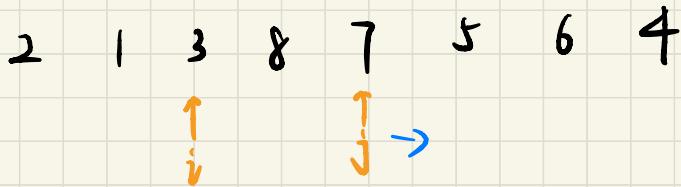
(这里都采用从小到大排序)

重点学习：快速排序、归并排序、堆排序

## ① 快速排序

分而治之  $\Rightarrow$  找出基线，不断将问题分解，直到符合基线条件





我们定义函数 `randomized_quicksort(nums, l, r)` 为对 `nums` 数组里  $[l, r]$  的部分进行排序，每次先调用 `randomized_partition` 函数对 `nums` 数组里  $[l, r]$  的部分进行划分，并返回分界值的下标 `pos`，然后按上述将的递归调用 `randomized_quicksort(nums, l, pos - 1)` 和 `randomized_quicksort(nums, pos + 1, r)` 即可。

那么核心就是划分函数的实现了，划分函数一开始需要确定一个分界值（我们称之为 主元 pivot），然后再进行划分。而主元的选取有很多种方式，这里我们采用随机的方式，对当前划分区间  $[l, r]$  里的数等概率随机一个作为我们的主元，再将主元放到区间末尾，进行划分。

整个划分函数 `partition` 主要涉及两个指针  $i$  和  $j$ ，一开始  $i = l - 1$ ， $j = l$ 。我们需要实时维护两个指针使得任意时候，对于任意数组下标  $k$ ，我们有如下条件成立：

1.  $l \leq k \leq i$  时， $nums[k] \leq pivot$ 。
2.  $i + 1 \leq k \leq j - 1$  时， $nums[k] > pivot$ 。
3.  $k == r$  时， $nums[k] = pivot$ 。

我们每次移动指针  $j$ ，(如果  $nums[j] > pivot$ ，我们只需要继续移动指针  $j$ ) 即能使上述三个条件成立，(否则我们需要将指针  $i$  加一，然后交换  $nums[i]$  和  $nums[j]$ ，再移动指针  $j$ ) 才能使得三个条件成立。

当  $j$  移动到  $r - 1$  时结束循环，此时我们可以由上述三个条件知道  $[l, i]$  的数都小于等于主元  $pivot$ ， $[i + 1, r - 1]$  的数都大于主元  $pivot$ ，那么我们只要交换  $nums[i + 1]$  和  $nums[r]$ ，即能使得  $[l, i + 1]$  区间的数都小于  $[i + 2, r]$  区间的数，完成一次划分，且分界值下标为  $i + 1$ ，返回即可。

```

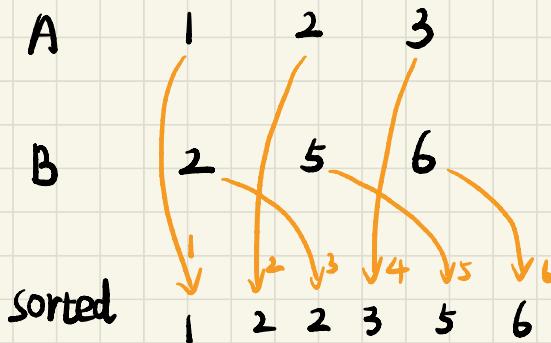
/*快速排序
vector<int> A1 = {6,3,2,7,1,5,8,4};
quicksort(A1, 0, (int)A1.size()-1);
*/
int partition(vector<int>& nums, int l, int r) {
    int i = rand() % (r - l + 1) + l; // 随机选一个作为我们的主元
    swap(nums[r], nums[i]);
    int pivot = nums[r];
    i = l - 1;
    for (int j = l; j <= r - 1; j++) {
        if (nums[j] <= pivot) {
            i = i + 1;
            swap(nums[i], nums[j]);
        }
    }
    swap(nums[i + 1], nums[r]);
    return i + 1;
}
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);
    }
}

```

\* 注意若是 string 类型，输入形参也要加 &

② 归并排序  $\Rightarrow$  最适用于外部排序

分而治之



## 思路

归并排序利用了分治的思想来对序列进行排序。对一个长为  $n$  的待排序的序列，我们将其分解成两个长度为  $\frac{n}{2}$  的子序列。每次先递归调用函数使两个子序列有序，然后我们再线性合并两个有序的子序列使整个序列有序。

## 算法

定义 `mergeSort(nums, 1, r)` 函数表示对 `nums` 数组里  $[l, r]$  的部分进行排序，整个函数流程如下：

1. 递归调用函数 `mergeSort(nums, 1, mid)` 对 `nums` 数组里  $[l, mid]$  部分进行排序。
2. 递归调用函数 `mergeSort(nums, mid + 1, r)` 对 `nums` 数组里  $[mid + 1, r]$  部分进行排序。
3. 此时 `nums` 数组里  $[l, mid]$  和  $[mid + 1, r]$  两个区间已经有序，我们对两个有序区间线性归并即可使 `nums` 数组里  $[l, r]$  的部分有序。

线性归并的过程并不难理解，由于两个区间均有序，所以我们维护两个指针  $i$  和  $j$  表示当前考虑到  $[l, mid]$  里的第  $i$  个位置和  $[mid + 1, r]$  的第  $j$  个位置。

如果 `nums[i] <= nums[j]`，那么我们就将 `nums[i]` 放入临时数组 `tmp` 中并让 `i += 1`，即指针往后移。否则我们就将 `nums[j]` 放入临时数组 `tmp` 中并让 `j += 1`。如果有一个指针已经移到了区间的末尾，那么就把另一个区间里的数按顺序加入 `tmp` 数组中即可。

这样能保证我们每次都是让两个区间中较小的数加入临时数组里，那么整个归并过程结束后  $[l, r]$  即为有序的。

```
/*归并排序
vector<int> A1 = {6,3,2,7,1,5,8,4};
tmp.resize((int)A1.size(), 0);
mergesort(A1, 0, (int)A1.size()-1);
*/
vector<int> tmp;
void mergesort(vector<int>& nums, int l, int r) {
    if (l >= r) return;
    int mid = (l + r) / 2; → 12
    mergesort(nums, l, mid);
    mergesort(nums, mid + 1, r);
    int i = l, j = mid + 1;
    int cnt = 0;
    while (i <= mid && j <= r) {
        if (nums[i] <= nums[j]) {tmp[cnt++] = nums[i++];}
        else {tmp[cnt++] = nums[j++];}
    }
    while (i <= mid) {tmp[cnt++] = nums[i++];}
    while (j <= r) {tmp[cnt++] = nums[j++];}
    for (int i = 0; i < r - l + 1; ++i) {
        nums[i + l] = tmp[i];
    }
}
```

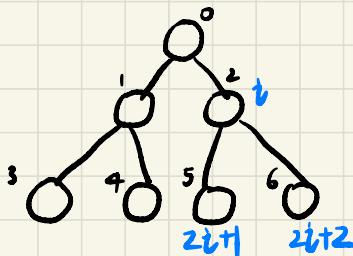
### ③ 堆排序

#### 预备知识

- 堆

#### 思路和算法

堆排序的思想就是先将待排序的序列建成大根堆，使得每个父节点的元素大于等于它的子节点。此时整个序列最大值即为堆顶元素，我们将其与末尾元素交换，使末尾元素为最大值，然后再调整堆顶元素使得剩下的 $n - 1$ 个元素仍为大根堆，再重复执行以上操作我们即能得到一个有序的序列。



$$\text{len} = 7 - 1 = 6$$

$\text{len}/2 = 3 \Rightarrow$  最左边的叶子节点

```
/*堆排序
vector<int> A1 = {6,3,2,7,1,5,8,4};
mergesort(A1);
*/
void maxHeapify(vector<int>& nums, int i, int len) { 相当于二叉树的调整 => 当前元素 i 的下沉
    for (; (i << 1) + 1 <= len;) {
        int lson = (i << 1) + 1; 左子节点
        int rson = (i << 1) + 2; 右子节点
        int large;
        if (lson <= len && nums[lson] > nums[i]) {large = lson;}
        else {large = i;}
        if (rson <= len && nums[rson] > nums[large]) {large = rson;}
        if (large != i) {
            swap(nums[i], nums[large]);
            i = large;
        } else {
            break;
        }
    }
}
void buildMaxHeap(vector<int>& nums, int len) { 初始化最大堆
    for (int i = len / 2; i >= 0; --i) {
        maxHeapify(nums, i, len);
    }
}
void heapSort(vector<int>& nums) {
    int len = (int)nums.size() - 1;
    buildMaxHeap(nums, len); ==> 将数组整理成堆(堆有序)，只保证父节点 > 子节点
    for (int i = len; i >= 1; --i) {
        swap(nums[i], nums[0]);
        len -= 1;
        maxHeapify(nums, 0, len); ==> 当前元素 0 的下沉，此循环同值在节点 i > 子节点，实现堆的真正有序
    }
}
```

## ④ 选择排序(了解)

### 贪心算法

减治思想 → 大而化小，小而化了，如二分查找

思路：每一轮选取未排定的部分中最小的部分交换到未排定部分的最开头，经过若干个步骤，就能排定整个数组。即：先选出最小的，再选出第 2 小的，以此类推。

### 数组：

```
/*选择排序
| selectsort(nums);
*/
void selectsort(vector<int>& nums){
    int len = nums.size();
    for(int i = 0; i<len-1; i++){
        int min = i;
        for(int j = i+1; j<len; j++){
            if(nums[j] < nums[min]) {min = j;}
        }
        if(min != i) {swap(nums[i], nums[min]);}
    }
}
```

### 单向链表：

```
ListNode* insertionSortList(ListNode* head) {
    ListNode* tmp;
    ListNode* tmp1;
    int val0;
    for(tmp = head; tmp != nullptr; tmp = tmp->next) {
        for(tmp1 = tmp->next; tmp1 != nullptr; tmp1 = tmp1->next) {
            if(tmp->val > tmp1->val) {
                val0 = tmp->val;
                tmp->val = tmp1->val;
                tmp1->val = val0;
            }
        }
    }
    return head;
}
```

## ⑤ 冒泡排序(了解)

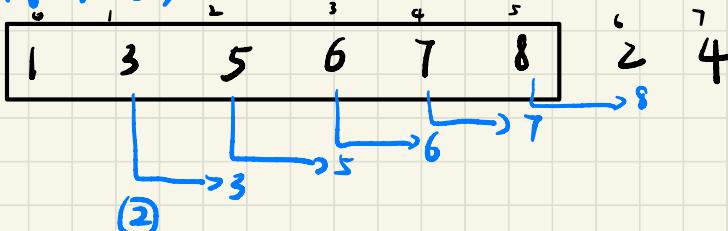
冒泡排序是最基本的排序算法，排序思路如下：

- 1.一边比较一边向后两两交换，将最大值冒泡到最后一位。
- 2.循环该过程n-1次(n为数组长度)，数组此时为升序排列。

```
/*起泡排序
| bubblesort(nums);
*/
void bubblesort(vector<int>& nums)
{
    int len = nums.size();
    for(int i = 0; i<len-1; i++){
        for(int j = 0; j<len-i-1; j++){
            if(nums[j] > nums[j+1]){
                swap(nums[j], nums[j+1]);
            }
        }
    }
}
```

6	2	1	8
2	6	1	8
2	1	6	8
1	2	6	8

## ⑥ 插入排序(熟悉)



$i=6$

$temp=2$

$j=6 \rightarrow 5 \rightarrow \dots \rightarrow 1$

### 数组:

假设一个数组，在其内部，数已经按照升序排列，此时有一个新的数  $a$  要加入数组，那么数组内大于  $a$  的数字需不断地向后腾出位置，直到  $a$  找到自己的位置，就可以将  $a$  插入该位置，此时原数组仍保持升序排列。

同理，插入排序就是将已排序部分当成一个小数组，未排序部分将一个一个插入到小数组当中，循环插入，直至排序完成。

### 双向链表: 147题

```
/*插入排序
| insertsort(nums);
*/
void insertsort(vector<int>& nums)
{
    int len = nums.size();
    for(int i = 1; i < len; i++){
        int temp = nums[i];
        int j = i;
        while(j > 0 && nums[j-1] > temp){
            nums[j] = nums[j-1];
            j--;
        }
        nums[j] = temp;
    }
}
```

```
ListNode* insertsortList(ListNode* head) {
    if(head == nullptr) {return head;}
    ListNode* dummy = new ListNode();
    dummy->next = head;
    ListNode* pre = dummy, *cur = head->next, *finalsort = head;
    for(;finalsort->next != nullptr;){
        if(finalsort->val < cur->val){
            finalsort = finalsort->next;
        } else{
            pre = dummy;
            while(cur->val > pre->next->val){
                pre = pre->next;
            }
            finalsort->next = cur->next;
            cur->next = pre->next;
            pre->next = cur;
        }
        cur = finalsort->next;
    }
    return dummy->next;
}
```

对于链表而言，插入元素时只需要更新相邻节点的指针即可。不需要像数组一样将插入位置后面的元素往移动。因此插入操作的时间复杂度是  $O(1)$ ，但是找到插入位置需要遍历链表中的节点，时间复杂度是  $O(n)$ ，因此链表插入排序的总时间复杂度仍然是  $O(n^2)$ ，其中  $n$  是链表的长度。

对于双向链表而言，只有指向后一个节点的指针，因此需要从链表的头节点开始往后遍历链表中的节点，寻找插入位置。

对链表进行插入排序的具体过程如下。

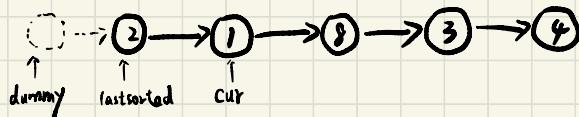
- 首先判断给定的链表是否为空。若为空，则不需要进行排序，直接返回。
- 创建虚节点  $dummyHead$ ，令  $dummyHead.next = head$ 。引入虚节点是为了便于在  $head$  节点之前插入节点。
- 维护  $lastSorted$  为链表的已排序部分的最后一个节点。初始时  $lastSorted = head$ 。
- 维护  $curr$  为待插入的元素。初始时  $curr = head.next$ 。
- 比较  $lastSorted$  和  $curr$  的节点值。
  - 若  $lastSorted.val \leq curr.val$ ，说明  $curr$  应该位于  $lastSorted$  之后，将  $lastSorted$  后移一位， $curr$  变成新的  $lastSorted$ 。
  - 否则，从链表的头节点开始往后遍历链表中的节点，寻找插入  $curr$  的位置。令  $prev$  为插入  $curr$  的位置的前一个节点。进行如下操作，完成对  $curr$  的插入：

```
lastSorted.next = curr;
curr.next = prev.next;
prev.next = curr;
```

6.令  $curr = lastSorted.next$ ，此时  $curr$  为下一个待插入的元素。

7.重复第 5 步和第 6 步，直到  $curr$  变成空，排序结束。

8.返回  $dummyHead.next$ ，为排序后的链表的头节点。



## ⑦ 希尔排序(分解)

$h=1 \quad 4 \quad 13 \quad 40 \quad 121 \dots$

希尔排序，即高级插入排序，是对插入排序的优化，思路如下：

- 将一个长数组按照相同的间隔  $h$  分为多个小数组，每个小数组分别进行插入排序。
- 将间隔  $h$  缩小，并继续排序，直至间隔为 1。

可以证明出当间隔  $h=3*h+1$  时，希尔排序平均时间复杂度最优，推理过程此处省略，有兴趣的读者可自行查阅。

```

/*希尔排序
    shellsort(nums);
*/
void shellsort(vector<int>& nums){
    int len = nums.size();
    int gap = 1;
    while(gap < len/3) {gap = gap*3+1;}
    while(gap>0){
        for(int i = gap; i<len; i++){
            int temp = nums[i];
            int j = i;
            while(j>=gap && nums[j-gap]>temp){
                nums[j] = nums[j-gap];
                j -= gap;
            }
            nums[j] = temp;
        }
        gap = gap/3;
    }
}

```

## ⑧ 桶排序(了解)

计数排序、基数排序是其特殊形式

排序方式	时间复杂度			空间复杂度	稳定性	复杂性
	平均情况	最坏情况	最好情况			
插入排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
希尔排序	$O(n^{1.3})$			$O(1)$	不稳定	较复杂
冒泡排序	$O(n^2)$	$O(n^2)$	$O(n)$	$O(1)$	稳定	简单
快速排序	$O(n \log_2 n)$	$O(n^2)$	$O(n \log_2 n)$	$O(\log_2 n)$	不稳定	较复杂
选择排序	$O(n^2)$	$O(n^2)$	$O(n^2)$	$O(1)$	不稳定	简单
堆排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(1)$	不稳定	较复杂
归并排序	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n \log_2 n)$	$O(n)$	稳定	较复杂
基数排序	$O(d(n+r))$	$O(d(n+r))$	$O(d(n+r))$	$O(r)$	稳定	较复杂

# 习题:

## 快速排序：

- 基础版本：对int数组或char数组排序。
- 提升版本：给定一些由【】或者空格或者[:]隔开的string字符串，将其中的内容按字典序排序，或者逆字典序排序。
- 剑指 Offer 45. 把数组排成最小的数**：输入一个非负整数数组，把数组里所有数字拼接起来排成一个数，打印能拼接出的所有数字中最小的一个。
- 剑指 Offer 40. 最小的k个数和面试题 17.14. 最小K个数**：找出数组中最小的k个数。以任意顺序返回这k个数均可。  
提升：按升序或者降序的顺序返回这k个数。
- 215. 数组中的第K个最大元素**：在未排序的数组中找到第 k 个最大的元素。请注意，你需要找的是数组排序后的第 k 个最大的元素，而不是第 k 个不同的元素。
- 75. 颜色分类**：给定一个包含红色、白色和蓝色，一共 n 个元素的数组，原地对它们进行排序，使得相同颜色的元素相邻，并按照红色、白色、蓝色顺序排列。

## 2. Leetcode 1153题：字符串排序

建立 vector<string> strarray;

也可用 swap、emplace\_back 等函数

字符串比较也可直接用 <、>、<=、>=

直接套用快排

string tmp = ""; // 初始化字符串

## 3. 更改比较大小的注释，快排代码可直接套用

- 若拼接字符串  $x + y > y + x$ ，则  $x$  “大于”  $y$ ；
- 反之，若  $x + y < y + x$ ，则  $x$  “小于”  $y$ ；

$x$  “小于”  $y$  代表：排序完成后，数组中  $x$  应在  $y$  左边；“大于” 则反之。

↳ 如  $\text{pivot} = 1$  时， $1 < 3 \& 1 < 3$

∴  $\text{pivot}$  在 3 的后面，所以应放到  $\text{pivot}$  后面

to\_string：整数转为字符串

## 堆排序：

以下题目是可以用堆排，但不仅限于用堆排。通常可以用堆排的题也可以用快排和桶排。

- 347. 前 K 个高频元素**：给定一个非空的整数数组，返回其中出现频率前 k 高的元素。
- 692. 前K个高频单词**：给一非空的单词列表，返回前 k 个出现次数最多的单词。
- 973. 最接近原点的 K 个点**：有一个由平面上的点组成的列表 points。需要从中找出 K 个距离原点 (0, 0) 最近的点。
- 1167. 连接棒材的最低费用**：会员题，可以用堆排。

## 1. 遍历vector几种方法：

① for(int i=0; i < v.size()-1; i++) {  
 cout << v[i];  
}

③ for(auto & vv : v) {  
 cout << vv;  
}

② vector<int>::iterator iter;  
for(iter = v.begin(); iter != v.end(); iter++) {  
 cout << (\*iter);  
}

}

## 使用 unordered\_map

<https://www.jianshu.com/p/56bb01df8ac7>

使用 `vector<pair<int, int>> v` ; 其中插入元素为 `v.emplace_back(make_pair(1, 2))` ;  
单例切片结构图为 `pair<int, int> p(1, 2)` ;

```
unordered_map<int, int> nums_map;
for(auto& n:nums) {
    nums_map[n]++;
}
vector<pair<int, int>> numsp;
for(auto& n:nums_map) {
    numsp.emplace_back(n);
}
```

也可使用优先队列 `priority_queue<int, vector<int>, greater<int> a;`  $\Rightarrow$  大根堆  
实现堆

`t = a.top();` 提取堆顶

`a.pop();` 弹出堆顶

`a.emplace(t);` 插入t并自动排序

<https://www.cnblogs.com/huashanqingzhu/p/11040390.html>

2、为了让堆排序严格按照从节点，左节点在节点，一定得执行堆顶元素放到最左位，然后依次循环  
`swap(nums[i], nums[0])`

找出前k个高频(最大)元素 题型，使用堆排序的做法：

在这里，我们可以利用堆的思想：建立一个小顶堆，然后遍历「出现次数数组」：

- 如果堆的元素个数小于  $k$ ，就可以直接插入堆中。
- 如果堆的元素个数等于  $k$ ，则检查堆顶与当前出现次数的大小。如果堆顶更大，说明至少有  $k$  个数字的出现次数比当前值大，故舍弃当前值；否则，就弹出堆顶，并将当前值插入堆中。

```
vector<pair<int, int>> numsheap;
for(int p = 0; p < numsp.size(); p++) {
    if(numsheap.size() < k) {
        numsheap.emplace_back(numsp[p]);
        buildminheap(numsheap, numsheap.size() - 1);
    }
    else {
        if(numsp[p].second > numsheap[0].second) {
            numsheap[0] = numsp[p];
            minheapify(numsheap, 0, numsheap.size() - 1);
        }
    }
}
```

另外，如果追加要求此  $k$  个元素由大到小排序输出，将抛弃堆顶到最后一堆的循环；若无限制则不用抛弃

## 1.3 并排序

1. 剑指 Offer 51. 数组中的逆序对：在数组中的两个数字，如果前面一个数字大于后面的数字，则这两个数字组成一个逆序对。输入一个数组，求出这个数组中的逆序对的总数。
2. 315. 计算右侧小于当前元素的个数：给定一个整数数组 `nums`，按要求返回一个新数组 `counts`。数组 `counts` 有该性质：`counts[i]` 的值是 `nums[i]` 右侧小于 `nums[i]` 的元素的数量。

# 1. 看到逆序对查找，则是用归并排序，并大幅缩短时间

有点类似于二分查找

```

while(i<=mid && j<=r){
    if(nums[i] <= nums[j]) {tmp[n++] = nums[i++];}
    else{
        tmp[n++] = nums[j++];
        cnt += mid - i + 1;
    }
}

```

只多了这样一行

二分查找  $\Rightarrow$

```

/*二分查找-找出x>=value的下界
A[9] = {0,1,2,3,3,3,4,5,6};
lower_bound1(A, 0, 8, 3);
*/
int lower_bound1(int array[], int first, int last, int value)
{
    while(first < last)
    {
        int mid = first + (last-first)/2;
        if(array[mid] < value) {first = mid + 1;}
        else {last = mid;}
    }
    return first;
}

```

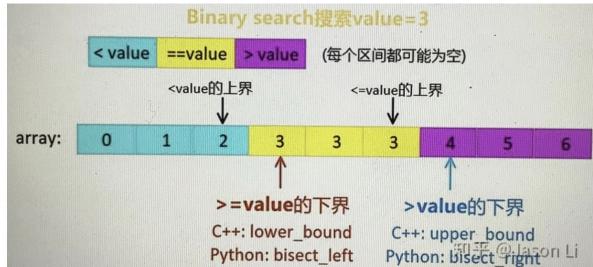
```

/*二分查找-找出x>value的下界
A[9] = {0,1,2,3,3,3,4,5,6};
upper_bound1(A, 0, 8, 3);
*/
int upper_bound1(int array[], int first, int last, int value)
{
    while(first < last)
    {
        int mid = first + (last-first)/2;
        if(array[mid] <= value) {first = mid + 1;}
        else {last = mid;}
    }
    return first;
}

```

lower\_bound(value) -1 找出 $x < value$ 的上界

upper\_bound(value)-1 找出 $x \leq value$ 上界



## 2. 降序的归并排序

可使用 tmpindex 和 index 来记录数组各元素最初的位置，

原理和 tmp 相似

初始化时  $\Rightarrow$

```
for(int c=0; c<nums.size(); c++) {index[c] = c;}
```

数组中元素位置发生变化  $\Rightarrow$

用 tmpindex 替换记录

最后更新 index  $\Rightarrow$

```

for(int n = 0; n<r-1; n++){
    nums[n+1] = tmp[n];
    index[n+1] = tmpindex[n];
}

```

# 桶排序

以上快排和堆排的题也都可以用桶排，这边列举的是一个用桶排比较方便的题目，其实题解区还有不用排序的做法。大家多看看不同题解，量力而行。

1. 1122. 数组的相对排序：给你两个数组，arr1 和 arr2，arr2 中的元素各不相同。arr2 中的每个元素都出现在 arr1 中对 arr1 中的元素进行排序，使 arr1 中项的相对顺序和 arr2 中的相对顺序相同。未在 arr2 中出现过的元素需要按照升序放在 arr1 的末尾。
2. 621. 任务调度器：给你一个用字符数组 tasks 表示的 CPU 需要执行的任务列表。其中每个字母表示一种不同种类的任务。任务可以以任意顺序执行，并且每个任务都可以在 1 个单位时间内执行完。在任何一个单位时间，CPU 可以完成一个任务，或者处于待命状态。然而，两个相同种类的任务之间必须有长度为整数 n 的冷却时间，因此至少有连续 n 个单位时间内 CPU 在执行不同的任务，或者在待命状态。你需要计算完成所有任务所需要的最短时间。

1、快排+二分查找 或 哈希表计数，桶的思想  
unordered\_map

2、快排+桶的思想 参考方法 <https://leetcode.cn/problems/task-scheduler/solution/tong-zi-by-popopop/>

排序算法在笔试面试中都是经常遇到的题目。在实际项目中，可能更多人选择直接用内库解决问题。但是在面试时候，面试官更感兴趣的是你对一些基础排序算法的理解，而不仅仅是简单的使用内库。就算你用内库很快的写完了，也许还会碰到面试官问你这个内库函数的基本原理是什么。我觉得面试者在面试前要尽可能的充分准备，多去思考一些深层次的东西，注重各大算法的基本原理和对比，而不是只用调用库函数，这样才能在面试中脱颖而出。

# 2. DFS(深度优先搜索)

## DFS ≈ 回溯算法

### 回溯算法与深度优先遍历

以下是维基百科中「回溯算法」和「深度优先遍历」的定义。

**回溯法** 采用试错的思想，它尝试分步的去解决一个问题。在分步解决问题的过程中，当它通过尝试发现现有的分步答案不能得到有效的正确的解答的时候，它将取消上一步甚至是上几步的计算，再通过其它的可能的分步解答再次尝试寻找问题的答案。回溯法通常用最简单的递归方法来实现，在反复重复上述的步骤后可能出现两种情况：

- 找到一个可能存在的正确的答案；
- 在尝试了所有可能的分步方法后宣告该问题没有答案。

**深度优先搜索 算法**（英语：Depth-First-Search，DFS）是一种用于遍历或搜索树或图的算法。这个算法会尽可能深的搜索树的分支。当结点  $v$  的所在边都已被探寻过，搜索将回溯到发现结点  $v$  的那条边的起始结点。这一过程一直进行到已发现从源结点可达的所有结点为止。如果还存在未被发现的结点，则选择其中一个作为源结点并重复以上过程，整个进程反复进行直到所有结点都被访问为止。

### 与动态规划的区别

#### 共同点

用于求解多阶段决策问题。多阶段决策问题即：

- 求解一个问题分为很多步骤（阶段）；
- 每一个步骤（阶段）可以有多种选择。

#### 不同点

- 动态规划只需求我们评估最优解是多少，最优解对应的具体解是什么并不要求。因此很适合应用于评估一个方案的效果；
- 回溯算法可以搜索得到所有的方案（当然包括最优解），但是本质上它是一种遍历算法，时间复杂度很高。

## 使用DFS遍历图

```
void DFS(int s) //DFS迭代版, [dTime,fTime]为活跃期, parent为父顶点*/  
{  
    int clock = 0; int v = s; stack<int> S;  
    do{  
        if(visited[v] == false)  
        {  
            S.push(v); dTime[v] = clock;  
            while(!S.empty())  
            {  
                v = S.top();  
                ++clock; visited[v] = true;  
                bool flag = false;  
                for(int i = 0; i < Graph[v].size(); i++)  
                {  
                    if(visited[Graph[v][i]] == false)  
                    {  
                        Graph[v][i].parent = v; S.push(Graph[v][i]);  
                        dtime[Graph[v][i]] = clock;  
                        flag = true;  
                        break;  
                    }  
                    flag = false;  
                }  
                if(!flag) {fTime[v] = clock; S.pop();}  
            }  
        }  
    }while(s != (v = (++v % n)));
```

```
void DFS(int s)//DFS递归版, [dTime,fTime]为活跃期, parent为父顶点  
//S为图的起点  
{  
    int clock = 0; int v = s;  
    do{  
        if(visited[v] == false)  
        {  
            dfs(v, clock);  
        }  
    }while(s != (v = (++v % n)));  
  
void dfs(int v, int& clock)  
{  
    dTime[v] = ++clock; visited[v] = true;  
    for(int i = 0; i < Graph[v].size(); i++)  
    {  
        if(visited[Graph[v][i]] == false)  
        {  
            Graph[v][i].parent = v;  
            dfs(Graph[v][i], clock);  
        }  
    }  
    fTime[v] = ++clock;  
}
```

## 全排序类题目

46. 全排列：给定一个 **没有重复** 数字的序列，返回其所有可能的全排列。
47. 全排列 II：给定一个 **有重复** 数字的序列，**按任意顺序** 返回所有不重复的全排列。
- 面试题 08.07. 无重复字符串的排列组合：计算某字符串的所有排列组合，字符串每个字符 **均不相同**。
- 面试题 08.08. 有重复字符串的排列组合：计算 **有重复** 字符串的所有排列组合，输出结果中不能有重复的字符串。
- 剑指 Offer 38. 字符串的排列：虽然题干不太一样，但是代码可以直接 **用上一题的**。
60. 排列序列：原名为第k个排序，难度较大，笔试更可能遇到。此题可以使用dfs+剪枝，推荐@liweiwei1419的题解：深度优先遍历 + 剪枝、有序数组模拟

7. 784. 字母大小写全排列：给定一个字符串S，通过将字符串S中的每个字母转变大小写获得一个新的字符串，返回所有可能得到的字符串集合。

**提升：**可以在本地编译环境试试将所有输出结果按 **字典序排列** 或者 **逆字典序排列**。

其他所有元素都是某元素的子节点



特点  
⇒ 因此dfs中的子因  
考虑所有其他元素

## 1.

```
vector<vector<int>> permute(vector<int>& nums) {
    vector<bool> visited(nums.size(), false);
    vector<int> path;
    vector<vector<int>> ans;
    dfs(nums, 0, visited, path, ans);
    return ans;
}

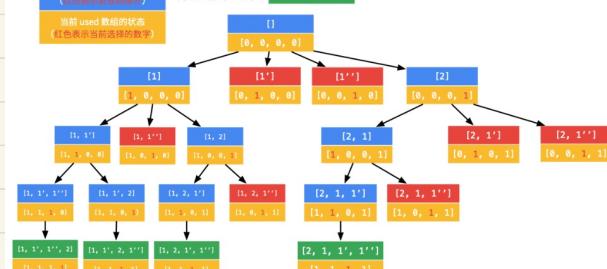
void dfs(vector<int>& nums, int depth, vector<bool>& visited, vector<int>& path, vector<vector<int>& ans){
    if(depth == nums.size()){
        ans.emplace_back(path);
        return;
    }
    for(int i = 0; i < nums.size(); i++){
        if(!visited[i]){
            path.emplace_back(nums[i]);
            visited[i] = true;
            dfs(nums, depth+1, visited, path, ans);
            visited[i] = false;
            path.pop_back();
        }
    }
    return;
}
```

## 2. 先排序，后在回溯中剪枝

图例：

- 当前得到的全排列的部分 (红色表示选择的数)
- 当前 used 数组的状态 (蓝色表示当前选择的数字)

构成全排列的数字：{1, 1', 1'', 2}



在for循环里添加的剪枝部分

```
if(visited[i] || (i > 0 && nums[i] == nums[i-1] && !visited[i-1])) {
    continue;
}
```

### 3. 与题1相同

string 加减字符

```

string S = "";
S += "a";
S.pop_back();

```

### 4. 5. 与题2相同

### 6.

按大小顺序列出所有排列情况，并一一标记，当  $n = 3$  时，所有排列如下：

1. "123"
2. "132"
3. "213"
4. "231"
5. "312"
6. "321"

给定  $n$  和  $k$ ，返回第  $k$  个排列。

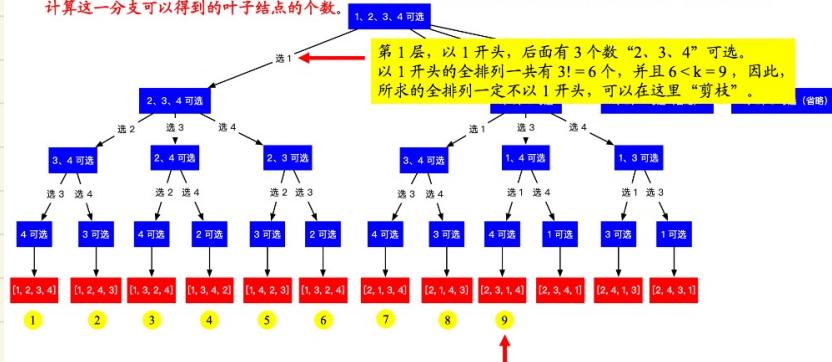
示例 1：

输入：  $n = 3, k = 3$   
输出： "213"

思路：

- 所求排列 **一定在叶子结点处得到**，进入每一个分支，可以根据已经选定的数的个数，进而计算还未选定的数的个数，然后计算阶乘，就知道这个分支的 **叶子结点** 的个数：
  - 如果  $k$  大于这一个分支将要产生的叶子结点数，直接跳过这个分支，这个操作叫「剪枝」；
  - 如果  $k$  小于等于这一个分支将要产生的叶子结点数，那说明所求的全排列一定在这一个分支将要产生的叶子结点里，需要递归求解。

输入： $n = 4, k = 9$ 。可以借助 LeetCode 第 46 题“全排列”的思想，依次得到全排列，输出第 9 个即可。  
**所求排列一定在叶子结点处得到。事实上，进入每一个分支的时候，我们都可以通过递归的层数，直接计算这一分支可以得到的叶子结点的个数。**



# 实现：

```

string getPermutation(int n, int k) {
    vector<int> nums;
    for(int p = 1; p <= n; p++){
        nums.emplace_back(p);
    }
    vector<bool> visited(nums.size(), false);
    vector<int> path;
    vector<int> factorial(nums.size());
    calculate_factorial(factorial, n);
    dfs_change(nums, 0, visited, path, n, k, factorial);
    string ans = "";
    for(int m = 0; m < n; m++){
        ans += to_string(path[m]);
    }
    return ans;
}

```

```

void dfs_change(vector<int>& nums, int index, vector<bool>& visited, vector<int>& path, int& n, int& k, vector<int>& factorial){
    if(index == n){
        return;
    }
    int cnt = factorial[n-1-index];
    for(int i = 0; i < nums.size(); i++){
        if(visited[i]) continue;
        if(cnt < k){
            k -= cnt;
            continue;
        }
        visited[i] = true;
        path.emplace_back(nums[i]);
        dfs_change(nums, index+1, visited, path, n, k, factorial);
        return;
    }
}

```

## 计算阶乘数组

```

void calculate_factorial(vector<int>& factorial, int n){
    factorial[0] = 1;
    for(int i = 1; i <= n; i++){
        factorial[i] = factorial[i-1] * i;
    }
}

```

7.

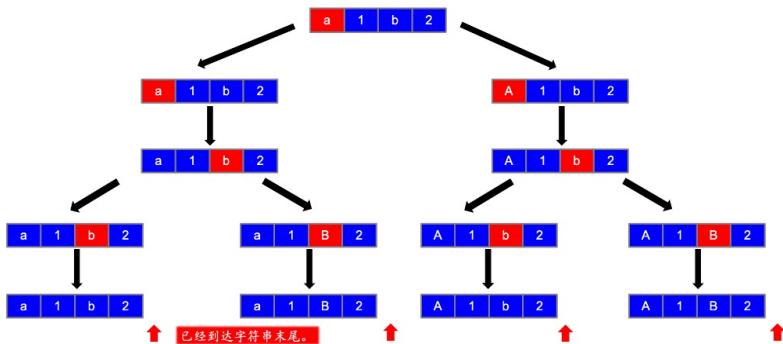
给定一个字符串 `s`，通过将字符串 `s` 中的每个字母转变大小写，我们可以获得一个新的字符串。

返回 所有可能得到的字符串集合。以 任意顺序 返回输出。

示例 1：

输入： `s = "a1b2"`  
输出： `["a1b2", "a1B2", "A1b2", "A1B2"]`

# 思路：



<https://leetcode.cn/problems/letter-case-permutation/solution/shen-du-you-xian-bian-li-hui-su-suan-fa-python-dai/>

# 实现：

```
vector<string> letterCasePermutation(string s) {
    vector<string> ans;
    dfs_change(s, 0, ans);
    return ans;
}

void dfs_change(string& s, int index, vector<string>& ans){
    if(index == s.size()){
        ans.emplace_back(s);
        return;
    }
    dfs_change(s, index+1, ans);
    if(s[index] >= 'A' && s[index] <= 'Z'){
        s[index] += 32;
        dfs_change(s, index+1, ans);
        s[index] -= 32;
    }
    if(s[index] >= 'a' && s[index] <= 'z'){
        s[index] -= 32;
        dfs_change(s, index+1, ans);
        s[index] += 32;
    }
}
```

字符'A'与'a'相差32，且'A'较小

## 组合类题目

17. 电话号码的字母组合：给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。答案可以按任意顺序返回。
77. 组合：给定两个整数 n 和 k，返回 1 ... n 中所有可能的k个数的组合。
39. 组合总和：给定一个 无重复元素 的数组 candidates 和一个目标数 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 无限制重复被选取。
40. 组合总和 II：给定一个数组 candidates（数组中的数字可能重复）和一个目标数 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。
216. 组合总和 III：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，并且每种组合中不存在重复的数字。
377. 组合总和 IV：给定一个由正整数组成且不存在重复数字的数组，找出和为给定目标正整数的组合的个数。PS：这题不用DFS，而是用动态规划，放在此处只是因为题目名字相似。



给定一个仅包含数字 2-9 的字符串，返回所有它能表示的字母组合。答案可以按任意顺序返回。

给出数字到字母的映射如下（与电话按键相同），注意 1 不对应任何字母。



示例 1：

```
输入: digits = "23"
输出: ["ad", "ae", "af", "bd", "be", "bf", "cd", "ce", "cf"]
```

每个元素都有自己的子节点。



因此树中的子树应考虑此时  
index下特有的子节点。

```

vector<string> letterCombinations(string digits) {
    unordered_map<char, string> phone_map{
        {'2', "abc"}, 
        {'3', "def"}, 
        {'4', "ghi"}, 
        {'5', "jkl"}, 
        {'6', "mno"}, 
        {'7', "pqrs"}, 
        {'8', "tuv"}, 
        {'9', "wxyz"} 
    };
    string path;
    vector<string> ans;
    if(digits.size() == 0) {return ans;}
    dfs(digits, 0, path, ans, phone_map);
    return ans;
}

```

```

void dfs(string& digits, int index, string& path, vector<string>& ans, unordered_map<char, string>& phone_map){
    if(index == digits.size()){
        ans.emplace_back(path);
        return;
    }
    for(int c = 0; c <(phone_map.at(digits[index])).size(); c++){
        path += (phone_map.at(digits[index]))[c];
        dfs(digits, index+1, path, ans, phone_map);
        path.pop_back();
    }
    return;
}

```

2、输入:  $n = 4, k = 2$

输出:

```

[
    [2,4],
    [3,4],
    [2,3],
    [1,2],
    [1,3],
    [1,4],
]

```

初始:

```

for(int i = index; i <n; i++) {
    path.emplace_back(nums[i]);
    dfs(nums, i+1, path, ans, k, n);
    path.pop_back();
}

```

|| 优化

```

for(int i = index; i <n - (k - path.size()) + 1; i++) {
    path.emplace_back(nums[i]);
    dfs(nums, i+1, path, ans, k, n);
    path.pop_back();
}

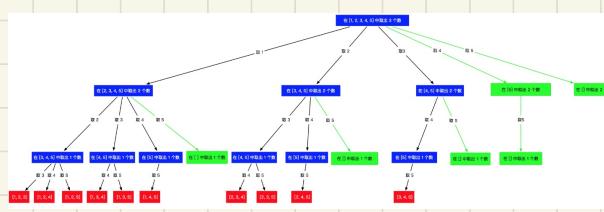
```

使用剪枝

事实上, 如果  $n = 7, k = 4$ , 从 5 开始搜索就已经没有意义了, 这是因为: 即使把 5 选上, 后面的数只有 6 和 7, 一共就 3 个候选数, 凑不出 4 个数的组合。因此, 搜索起点有上界, 这个上界是多少, 可以举几个例子分析。

分析搜索起点的上界, 其实是在深度优先遍历的过程中剪枝, 剪枝可以避免不必要的遍历, 剪枝剪得好, 可以大幅度节约算法的执行时间。

下面的图片绿色部分是剪掉的枝叶, 当  $n$  很大的时候, 能少遍历很多结点, 节约了时间。



例如:  $n = 6, k = 4$ 。

$\text{path.size()} = 1$  的时候, 接下来要选择 3 个数, 搜索起点最大是 4, 最后一个被选的组合是 [4, 5, 6] ;

$\text{path.size()} = 2$  的时候, 接下来要选择 2 个数, 搜索起点最大是 5, 最后一个被选的组合是 [5, 6] ;

$\text{path.size()} = 3$  的时候, 接下来要选择 1 个数, 搜索起点最大是 6, 最后一个被选的组合是 [6] ;

可以归纳出:

搜索起点的上界 + 接下来要选择的元素个数 - 1 = n

其中, 接下来要选择的元素个数 =  $k - \text{path.size()}$ , 整理得到:

搜索起点的上界 =  $n - (k - \text{path.size}) + 1$

所以, 我们的剪枝过程就是: 把  $i \leq n$  改成  $i \leq n - (k - \text{path.size}) + 1$  :

< <

3.

输入: candidates = [2,3,6,7], target = 7  
 输出: [[2,2,3],[7]]  
 解释:  
 2 和 3 可以形成一组候选， $2 + 2 + 3 = 7$ 。注意 2 可以使用多次。  
 7 也是一个候选， $7 = 7$ 。  
 仅有这两种组合。

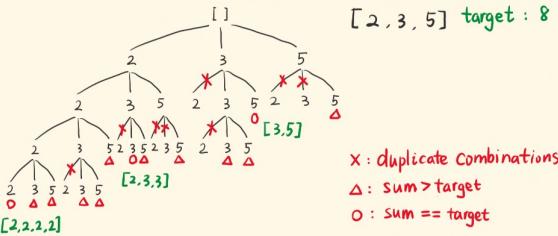
初步:

```
if(sum > target) {return;}
for(int i = index; i < candidates.size(); i++) {
    sum += candidates[i];
    path.emplace_back(candidates[i]);
    dfs(candidates, i, path, ans, target, sum);
    sum -= candidates[i];
    path.pop_back();
}
```

优化

排序后剪枝:

- ×: 当前组合和之前生成的组合重复了。
- △: 当前求和 > target, 不能选下去了, 返回。
- : 求和正好 == target, 加入解集, 并返回。



Tips:

什么时候使用 used 数组, 什么时候使用 index 变量

有些朋友可能会疑惑什么时候使用 used 数组, 什么时候使用 index 变量。这里为大家简单总结一下:

- 排列问题, 讲究顺序 (即 [2, 2, 3] 与 [2, 3, 2] 视为不同列表时), 需要记录哪些数字已经使用过, 此时用 used 数组;
- 组合问题, 不讲究顺序 (即 [2, 2, 3] 与 [2, 3, 2] 视为相同列表时), 需要按照某种顺序搜索, 此时使用 index 变量。

index

4.

输入: candidates = [10,1,2,7,6,1,5], target = 8,  
 输出:  
 [  
 [1,1,6],  
 [1,2,5],  
 [1,7],  
 [2,6]  
 ]

排序后剪枝

```
for(int i = index; i < candidates.size(); i++) {
    if(i < index && candidates[i] == candidates[i-1]) {continue;}
    if((sum + candidates[i]) > target) {break;}
    sum += candidates[i];
    path.emplace_back(candidates[i]);
    dfs(candidates, i+1, path, ans, target, sum);
    sum -= candidates[i];
    path.pop_back();
}
```

5.

输入:  $k = 3, n = 9$   
 输出:  $[[1,2,6], [1,3,5], [2,3,4]]$   
 解释:  
 $1 + 2 + 6 = 9$   
 $1 + 3 + 5 = 9$   
 $2 + 3 + 4 = 9$   
 没有其他符合的组合了。

## 题2与题3剪枝的结合

```
if(sum == n && path.size() == k){
    ans.emplace_back(path);
    return;
}
for(int i = index; i < nums.size() - (k - path.size()) + 1; i++){
    if((sum + nums[i]) > n) {break;} 剪枝1
    sum += nums[i];
    path.emplace_back(nums[i]);
    dfs(nums, i+1, path, ans, k, n, sum);
    sum -= nums[i];
    path.pop_back();
}
```

6.

输入:  $\text{nums} = [1,2,3]$ ,  $\text{target} = 4$

输出: 7

解释:

所有可能的组合为:

(1, 1, 1, 1)  
 (1, 1, 2)  
 (1, 2, 1)  
 (1, 3)  
 (2, 1, 1)  
 (2, 2)  
 (3, 1)

请注意, 顺序不同的序列被视作不同的组合。

## 使用DFS+剪枝

```
int combinationSum4(vector<int>& nums, int target) {
    sort(nums.begin(), nums.end());
    int cnt = 0;
    dfs(nums, 0, cnt, target, 0);
    return cnt;
}

void dfs(vector<int>& nums, int index, int& cnt, int target, int sum){
    if(sum == target){
        cnt++;
        return;
    }
    for(int i = 0; i < nums.size(); i++){
        if((sum + nums[i]) > target) {break;}
        sum += nums[i];
        dfs(nums, i, cnt, target, sum);
        sum -= nums[i];
    }
}
```

会超出时间，应该用动态规划(背包问题)

## 子集类题目

1. 78. 子集：给定一组 **不含重复元素** 的整数数组 nums，返回该数组所有可能的子集（幂集）。

2. 90. 子集 II：给定一个 **可能包含重复元素** 的整数数组 nums，返回该数组所有可能的子集（幂集）。

3. 面试题 08.04 幂集 跟上一题除了函数名不一样以外，其余代码可以一模一样。

**提升：**可以在本地编程环境试试将整数数组改成**字符串数组**，并且最后的结果按任意、字典序或者逆字典序顺序排列。

特点  
→ 每一次进入递归的 DFS 不用终止条件即可立即输出

1.

输入：nums = [1,2,3]  
输出：[[],[1],[2],[1,2],[3],[1,3],[2,3],[1,2,3]]

```
void dfs(vector<int>& nums, int index, vector<int>& path, vector<vector<int>>& ans){  
    ans.emplace_back(path);  
    for(int i = index; i<nums.size(); i++){  
        path.emplace_back(nums[i]);  
        dfs(nums, i+1, path, ans);  
        path.pop_back();  
    }  
    return;  
}
```

2.

输入：nums = [1,2,2]  
输出：[[],[1],[1,2],[1,2,2],[2],[2,2]]

排序后剪枝（与组合类题4相似）

```
for(int i = index; i<nums.size(); i++){  
    if(i > index && nums[i]==nums[i-1]) {continue;}  
    path.emplace_back(nums[i]);  
    dfs(nums, i+1, path, ans);  
    path.pop_back();  
}
```

3. 与题1相同

## 单词搜索类题目

1. 79. 单词搜索：给定一个二维网格和一个单词，找出该单词是否存在于网格中。单词必须按照字母顺序，通过相邻的单元格内的字母构成，其中“相邻”单元格是那些水平相邻或垂直相邻的单元格。同一个单元格内的字母不允许被重复使用。

2. 动画 Offer 12. 矩阵中的路径：别看此题题干好多字的样子，实际上代码和上一题一模一样，连函数名都没变。

3. 212. 单词搜索 II：个人觉得难度不适合普通面试者的面试，但是笔试时候很有可能考到。（本人遇到2-3次与此题类似的笔试题）

A	B	C	E
S	F	C	S
A	D	E	E

输入: board = [[“A”, “B”, “C”, “E”], [“S”, “F”, “C”, “S”], [“A”, “D”, “E”, “E”]], word = “ABCCED”  
输出: true

```
bool exist(vector<vector<char>>& board, string word) {
    vector<vector<bool>> visited(board.size(), vector<bool>(board[0].size(), false));
    vector<vector<int>> direction = {{1,0}, {-1,0}, {0,1}, {0,-1}}; → 方向 二维vector初始化
    for(int i = 0; i<board.size(); i++){
        for(int j = 0; j<board[0].size(); j++){ 从不同的起点开始出发
            if(dfs(board, i, j, 0, word, visited, direction)) {return true;}
        }
    }
    return false;
}

bool dfs(vector<vector<char>>& board, int x, int y, int index, string& word, vector<vector<bool>>& visited, vector<vector<int>>& direction){
    if(index == word.size()-1){
        return word[index] == board[x][y];
    }
    if(word[index] == board[x][y]){
        visited[x][y] = true;
        for(int i = 0; i<4; i++){
            int new_x = x + direction[i][0];
            int new_y = y + direction[i][1];
            if(new_x<=0 && new_x<board.size() && new_y<0 && new_y<board[0].size()){
                if(visited[new_x][new_y]) {continue;}
                if(dfs(board, new_x, new_y, index+1, word, visited, direction)) {return true;}
            }
        }
        visited[x][y] = false;
    }
    return false;
}
```

↑ 防止越界

## 2. 与题1相同

## 3. 回溯十字典树(前缀树)

o	a	a	n
e	t	a	e
i	h	k	r
i	f	l	v

输入: board = [[“o”, “a”, “a”, “n”], [“e”, “t”, “a”, “e”], [“i”, “h”, “k”, “r”], [“i”, “f”, “l”, “v”]], words = [“oath”, “pea”, “eat”, “rain”]  
输出: [“eat”, “oath”]

# 引入字典树用于快速查找字符串

前缀树（字典树）是一种树形数据结构，用于高效地存储和检索字符串数据集中的键。前缀树可以用  $O(|S|)$

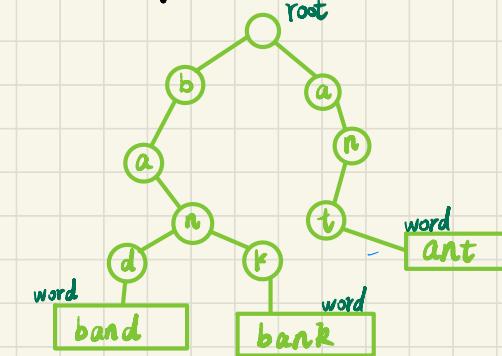
的时间复杂度完成如下操作，其中  $|S|$  是插入字符串或查询前缀的长度：

- 向前缀树中插入字符串  $word$ ；
- 查询前缀串  $prefix$  是否为已经插入到前缀树中的任意一个字符串  $word$  的前缀；

```
struct TrieNode{  
    string word;  
    unordered_map<char, TrieNode*> children;  
    TrieNode() {  
        this->word = "";  
    }  
};
```

```
void insertTrie(TrieNode* root, string& word){  
    TrieNode* node = root;  
    for(auto& c:word){  
        if(!node->children.count(c)){  
            node->children[c] = new TrieNode();  
        }  
        node = node->children[c];  
    }  
    node->word = word;  
}
```

如字符串 band、bank、ant 的存储



# 引入set数据结构用于去重

- 因为同一个单词可能在多个不同的路径中出现，所以我们需要使用哈希集合对结果集去重。
- 在回溯的过程中，我们不需要每一步都判断完整的当前路径是否是  $words$  中任意一个单词的前缀；而是可以记录下路径中每个单元格所对应的前缀树结点，每次只需要判断新增单元格的字母是否是上一个单元格对应前缀树结点的子结点即可。

[https://blog.csdn.net/weixin\\_52115456/article/details/124210086](https://blog.csdn.net/weixin_52115456/article/details/124210086)

```
vector<string> findWords(vector<vector<char>>& board, vector<string>& words) {  
    TrieNode* root = new TrieNode();  
    for(auto& word:words){  
        insertTrie(root, word);  
    }  
    vector<vector<bool>> visited(board.size(), vector<bool>(board[0].size(), false));  
    vector<vector<int>> direction = {{1,0}, {-1,0}, {0,1}, {0,-1}};  
    set<string> res;  
    for(int i = 0; i<board.size(); i++){  
        for(int j = 0; j<board[0].size(); j++){  
            dfs(board, i, j, root, visited, direction, res);  
        }  
    }  
    vector<string> ans;  
    for(auto& word:res){  
        ans.emplace_back(word);  
    }  
    return ans;  
}
```

```

void dfs(vector<vector<char>>& board, int x, int y, TrieNode* root, vector<vector<bool>>& visited,
vector<vector<int>>& direction, set<string>& res){
    if(!root->children.count(board[x][y])) {return;}
    root = root->children[board[x][y]];
    if(root->word.size() > 0) {res.insert(root->word);}
    visited[x][y] = true;
    for(int i = 0; i < 4; i++){
        int new_x = x + direction[i][0];
        int new_y = y + direction[i][1];
        if(new_x >= 0 && new_x < board.size() && new_y >= 0 && new_y < board[0].size()){
            if(visited[new_x][new_y]) {continue;}
            dfs(board, new_x, new_y, root, visited, direction, res);
        }
    }
    visited[x][y] = false;
}
};


```

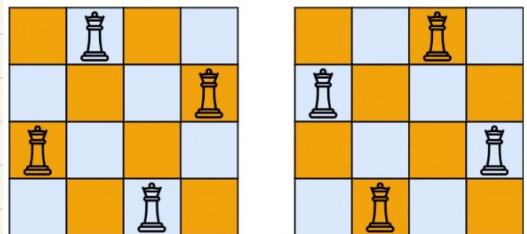
## N皇后问题

按照国际象棋的规则，皇后可以攻击与之处在同一行或同一列或同一斜线上的棋子。

n 皇后问题 研究的是如何将 n 个皇后放置在 n×n 的棋盘上，并且使皇后彼此之间不能相互攻击。

给你一个整数 n，返回所有不同的 n 皇后问题 的解决方案。

每一种解法包含一个不同的 n 皇后问题 的棋子放置方案，该方案中 'Q' 和 '.' 分别代表了皇后和空位。



输入: n = 4

输出: [["Q..","...Q","Q...","..Q."],  
["..Q.","Q...","...Q",".Q.."]]

解释: 如上图所示，4 皇后问题存在两个不同的解法。

```

vector<vector<string>> solveNQueens(int n) {
    string chess_row;
    for(int r = 0; r < n; r++) {chess_row += '.';}
    vector<string> chessboard(n, chess_row); 盒
    vector<vector<string>> ans;
    dfs(n, 0, chessboard, ans);
    return ans;
}

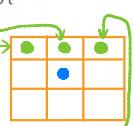
```

```

void dfs(int n, int row, vector<string>& chessboard, vector<vector<string>>& ans){
    if(row == n){ ⇒ 遍历完所有行且都有效，即可输出
        ans.emplace_back(chessboard);
        return;
    }
    for(int col = 0; col < n; col++){
        if(isValid(n, row, col, chessboard)){ → 判断此行列是否有效
            chessboard[row][col] = 'Q';
            dfs(n, row+1, chessboard, ans);
            chessboard[row][col] = '.';
        }
    }
    return;
}

bool isValid(int n, int row, int col, vector<string>& chessboard){
    for(int i = 0; i < row; i++){
        if(chessboard[i][col] == 'Q') {return false;}
    }
    for(int i = row-1, j = col-1; i >= 0 && j >= 0; i--, j--){
        if(chessboard[i][j] == 'Q') {return false;}
    }
    for(int i = row-1, j = col+1; i >= 0 && j < n; i--, j++){
        if(chessboard[i][j] == 'Q') {return false;}
    }
    return true;
}

```



# 3. BFS(宽度优先搜索)

本篇中的题目只是说可以利用BFS (Breadth First Search, 广度优先搜索) 的方法解决，而不是只能用这个方法解决。在某些大佬的题解中，甚至可以用三四种方式解决这个类型的题目，有兴趣的可以多翻一翻题解区高阅读量高赞或者标注为精选的题解哈。其实BFS在面试中出现的频率不如链表和树之类的知识点来的高，因为写清楚需要较长时间，但某些面试官会问面试者这类题目的思路是什么样的，所以理解此类题目的思路是非常重要的，起码自己回答的时候需要列点回答。

关于面试者的提前准备工作，建议视自身情况来定，如果说时间不够或者不能完全掌握所有知识点的，建议择其一记忆即可（选自己能理解的方法记忆，而不是一味选择最优解记忆）。学有余力的同学可以看看多种思路，开拓思维方式，方便自己在面试官面前装逼（注意别翻车就行）。

## 使用BFS遍历图

```
void BFS(int s)//BFS迭代版
{
    int clock = 0; int v = s; queue<int> Q;
    do{
        if(visited[v] == false)
        {
            visited[v] = true; Q.push(v);
            while(!Q.empty())
            {
                int v = Q.front(); Q.pop();
                dTime[v] = ++clock;
                for(int i = 0; i < Graph[v].size(); i++)
                {
                    if(visited[Graph[v][i]] == false)
                    {
                        visited[Graph[v][i]] = true;
                        Q.push(Graph[v][i]);
                        Graph[v][i].parent = v;
                    }
                }
                fTime[v] = ++clock;
            }
        }
    }while(s != (v = (++v % n)));
}
```

### 给定网格的题目

方法很多，例如：DFS、BFS和并查集。

130. 被围绕的区域：给定一个二维的矩阵，包含 'X' 和 'O' (字母 O)。找到所有被 'X' 围绕的区域，并将这些区域里所有的 'O' 用 'X' 填充。

这题可以用的方法很多，常见的有DFS、BFS和并查集，其中并查集这种数据结构看似不经常用，实际上分析此类题目时候会发挥比较好的作用。

200. 岛屿数量：给你一个由 '1' (陆地) 和 '0' (水) 组成的二维网格，请你计算网格中岛屿的数量。

同样也是可以用多种方法的一题。**提升：**可以思考下如果输入的不是char，而是int的话，该怎么处理。

695. 岛屿的最大面积：给定一个包含了一些 0 和 1 的非空二维数组grid。一个岛屿是由一些相邻的1(代表土地)构成的组合，这里的「相邻」要求两个1必须在水平或者竖直方向上相邻。你可以假设grid的四个边缘都被0 (代表水) 包围着。找到给定的二维数组中最大的岛屿面积。

- 面试题 16.19. 水域大小：你有一个用于表示一片土地的整数矩阵land，该矩阵中每个点的值代表对应地点的海拔高度。若值为0则表示水域。由垂直、水平或对角连接的水域为池塘。池塘的大小是指相连接的水域的个数。编写一个方法来计算矩阵中所有池塘的大小，返回值需要从小到大排序。

**提升：**做完可以与上一题对比下异同点。

1162. 地图分析：你现在手里有一份大小为NxN的网格grid，上面的每个单元格都用0和1标记好了。其中0代表海洋，1代表陆地，请你找出一个海洋单元格，这个海洋单元格到离它最近的陆地单元格的距离是最大的。

994. 腐烂的橘子：每分钟，任何与腐烂的橘子（在 4 个正方向上）相邻的新鲜橘子都会腐烂。返回直到单元格中没有新鲜橘子为止所必须经过的最短分钟数。

# 1、关键>寻找与边界联通的O

X	X	X	X
X	O	O	X
X	X	O	X
X	O	X	X



X	X	X	X
X	X	X	X
X	X	X	X
X	O	X	X

输入: board = [["X", "X", "X", "X"], ["X", "O", "O", "X"], ["X", "X", "O", "X"], ["X", "O", "X", "X"]],

输出: [[“X”, “X”, “X”, “X”], [“X”, “X”, “X”, “X”],

[“X”, “X”, “X”, “X”], [“X”, “O”, “X”, “X”]]

解释: 被包围的区间不会存在于边界上, 换句话说, 任何边界上的 ‘O’ 都不会被填充为 ‘X’。任何不在边界上, 或不与边界上的 ‘O’ 相连的 ‘O’ 最终都会被填充为 ‘X’。如果两个元素在水平或垂直方向相邻, 则称它们是“相连”的。

## ① DFS

```
class Solution {
public:
    void solve(vector<vector<char>>& board) {
        vector<vector<int>> direction = {{1,0}, {-1,0}, {0,1}, {0,-1}};
        for(int i = 0; i<board.size(); i++){
            for(int j = 0; j<board[0].size(); j++){
                if((i == 0 || j==0 || i==board.size()-1 || j == board[0].size()-1) && board[i][j] == 'O'){
                    dfs(board, i, j, direction);
                }
            }
        }
        for(int i = 0; i<board.size(); i++){
            for(int j = 0; j<board[0].size(); j++){
                if(board[i][j] == 'O') (board[i][j] = 'X');
                else if(board[i][j] == '#') (board[i][j] = 'O');
            }
        }
    }

    void dfs(vector<vector<char>>& board, int x, int y, vector<vector<int>>& direction){
        board[x][y] = '#';
        for(int i = 0; i<4; i++){
            int new_x = x + direction[i][0];
            int new_y = y + direction[i][1];
            if(new_x<0 && new_x<board.size() && new_y<0 && new_y<board[0].size()){
                if(board[new_x][new_y] == '#' || board[new_x][new_y] == 'X') {continue;}
                vector<int> pos = {new_x, new_y};
                q_bfs.push(pos);
                board[new_x][new_y] = '#';
            }
        }
    }
};

void solve(vector<vector<char>>& board) {
    queue<vector<int>> q_bfs;
    vector<vector<int>> direction = {{1,0}, {-1,0}, {0,1}, {0,-1}};
    for(int i = 0; i<board.size(); i++){
        for(int j = 0; j<board[0].size(); j++){
            if((i == 0 || j==0 || i==board.size()-1 || j == board[0].size()-1) && board[i][j] == 'O'){
                bfs(board, i, j, q_bfs, direction);
            }
        }
    }
    for(int i = 0; i<board.size(); i++){
        for(int j = 0; j<board[0].size(); j++){
            if(board[i][j] == 'O') (board[i][j] = 'X');
            else if(board[i][j] == '#') (board[i][j] = 'O');
        }
    }
}
```

## ② BFS

```
void bfs(vector<vector<char>>& board, int x, int y, queue<vector<int>>& q_bfs, vector<vector<int>>& direction) {
    board[x][y] = '#';
    vector<int> pos = {x, y};
    q_bfs.push(pos);
    while(!q_bfs.empty()){
        x = q_bfs.front()[0];
        y = q_bfs.front()[1];
        q_bfs.pop();
        for(int i = 0; i<4; i++){
            int new_x = x + direction[i][0];
            int new_y = y + direction[i][1];
            if(new_x<0 && new_x<board.size() && new_y<0 && new_y<board[0].size()){
                if(board[new_x][new_y] == '#' || board[new_x][new_y] == 'X') {continue;}
                vector<int> pos = {new_x, new_y};
                q_bfs.push(pos);
                board[new_x][new_y] = '#';
            }
        }
    }
}

void solve(vector<vector<char>>& board) {
    queue<vector<int>> q_bfs;
    vector<vector<int>> direction = {{1,0}, {-1,0}, {0,1}, {0,-1}};
    for(int i = 0; i<board.size(); i++){
        for(int j = 0; j<board[0].size(); j++){
            if((i == 0 || j==0 || i==board.size()-1 || j == board[0].size()-1) && board[i][j] == 'O'){
                bfs(board, i, j, q_bfs, direction);
            }
        }
    }
    for(int i = 0; i<board.size(); i++){
        for(int j = 0; j<board[0].size(); j++){
            if(board[i][j] == 'O') (board[i][j] = 'X');
            else if(board[i][j] == '#') (board[i][j] = 'O');
        }
    }
}
```

# ③ 并查集

并查集这种数据结构好像大家不太常用，实际上很有用，我在实际的 production code 中用过并查集。并查集常用来解决连通性的问题，即将一个图中连通的部分划分出来。当我们判断图中两个点之间是否存在路径时，就可以根据判断他们是否在一个连通区域。而道题我们其实求解的就是和边界的 O 在一个连通区域的问题。

并查集的思想就是：同一个连通区域内的所有点的根节点是同一个。将每个点映射成一个数字。先假设每个点的根节点就是他们自己，然后我们以此输入连通的点对，然后将其中一个点的根节点赋成另一个节点的根节点。这样这两个点所在连通区域又相互连通了。

并查集的主要操作有：

- find(int m)：这是并查集的基本操作，查找 m 的根节点。
- isConnected(int m, int n)：判断 m, n 两个点是否在一个连通区域。
- union(int m, int n)：合并 m, n 两个点所在的连通区域。

<https://leetcode.cn/problems/surrounded-regions/solution/bfsdi-gui-dfsfei-di-gui-dfsbing-cha-ji-by-ac-pipe/>

```
struct UnionFind{  
    vector<int> parents;  
    UnionFind(int totalnodes) {  
        parents.resize(totalnodes);  
        for(int i = 0; i < totalnodes; i++) {  
            parents[i] = i;  
        }  
    }  
};  
  
int find_parent(UnionFind* uf, int node) {  
    while(uf->parents[node] != node) {  
        uf->parents[node] = uf->parents[uf->parents[node]];  
        node = uf->parents[node];  
    }  
    return node;  
}  
  
void union_nodes(UnionFind* uf, int node1, int node2) {  
    int root1 = find_parent(uf, node1);  
    int root2 = find_parent(uf, node2);  
    if(root1 != root2) {  
        uf->parents[root2] = root1;  
    }  
}  
  
bool isdisconnected(UnionFind* uf, int node1, int node2) {  
    return find_parent(uf, node1) == find_parent(uf, node2);  
}
```

我们的思路是把所有边界上的 O 看做一个连通区域。遇到 O 就执行并查集合并操作，这样所有的 O 就会被分成两类

- 和边界上的 O 在一个连通区域内的。这些 O 我们保留。
- 不和边界上的 O 在一个连通区域内的。这些 O 就是被包围的，替换。

由于并查集我们一般用一维数组来记录，方便查找 parents，所以我们将二维坐标用 node 函数转化为一维坐标。

```
void solve(vector<vector<char>>& board) {  
    UnionFind* uf = new UnionFind(board.size()*board[0].size() + 1);  
    int dummynode = board.size()*board[0].size(); // 虚拟节点，边界上的O的父节点都是这个虚拟节点  
    for(int i = 0; i < board.size(); i++) {  
        for(int j = 0; j < board[0].size(); j++) {  
            if(board[i][j] == 'O') {  
                if(i == 0 || j == 0 || i == board.size() - 1 || j == board[0].size() - 1) {  
                    union_nodes(uf, i*board[0].size() + j, dummynode); // 和dummyNode 在一个连通区域的，那么就是0  
                }  
                else {  
                    if(i > 0 && board[i - 1][j] == 'O')  
                        union_nodes(uf, i*board[0].size() + j, (i - 1)*board[0].size() + j);  
                    if(i < board.size() - 1 && board[i + 1][j] == 'O')  
                        union_nodes(uf, i*board[0].size() + j, (i + 1)*board[0].size() + j);  
                    if(j > 0 && board[i][j - 1] == 'O')  
                        union_nodes(uf, i*board[0].size() + j, i*board[0].size() + (j - 1));  
                    if(j < board[0].size() - 1 && board[i][j + 1] == 'O')  
                        union_nodes(uf, i*board[0].size() + j, i*board[0].size() + (j + 1));  
                }  
            }  
        }  
    }  
    for(int i = 0; i < board.size(); i++) {  
        for(int j = 0; j < board[0].size(); j++) {  
            if(isdisconnected(uf, i*board[0].size() + j, dummynode)) {board[i][j] = '0';}  
            else {board[i][j] = 'X';}  
        }  
    }  
}
```

## 2.

给你一个由 '1' (陆地) 和 '0' (水) 组成的二维网格, 请你计算网格中岛屿的数量。

岛屿总是被水包围, 并且每座岛屿只能由水平方向和/或竖直方向上相邻的陆地连接形成。

此外, 你可以假设该网格的四条边均被水包围。

```
输入: grid = [
    ["1","1","1","1","0"],
    ["1","1","0","1","0"],
    ["1","1","0","0","0"],
    ["0","0","0","0","0"]
]
输出: 1
```

### ①BFS

```
int numIslands(vector<vector<char>>& grid) {
    vector<vector<bool>> visited(grid.size(), vector<bool>(grid[0].size(), false));
    queue<vector<int>> q_bfs;
    vector<vector<int>> direction = {{1, 0}, {-1, 0}, {0, 1}, {0, -1}};
    int cnt = 0;
    for(int i = 0; i < grid.size(); i++) {
        for(int j = 0; j < grid[0].size(); j++) {
            if(visited[i][j] && grid[i][j] == '1') {
                bfs(grid, i, j, visited, q_bfs, direction);
                cnt++;
            }
        }
    }
    return cnt;
}
```

```
void bfs(vector<vector<char>>& grid, int x, int y, vector<vector<bool>>& visited, queue<vector<int>>& q_bfs, vector<vector<int>> direction) {
    visited[x][y] = true;
    vector<int> pos = {x, y};
    q_bfs.push(pos);
    while(!q_bfs.empty()){
        x = q_bfs.front()[0];
        y = q_bfs.front()[1];
        q_bfs.pop();
        for(int i = 0; i < 4; i++){
            int new_x = x + direction[i][0];
            int new_y = y + direction[i][1];
            if(new_x > 0 && new_x < grid.size() && new_y > 0 && new_y < grid[0].size() && !visited[new_x][new_y]){
                if(grid[new_x][new_y] == '0' || visited[new_x][new_y] == true) {continue;}
                vector<int> pos = {new_x, new_y};
                q_bfs.push(pos);
                visited[new_x][new_y] = true;
            }
        }
    }
}
```

**关键 => 寻找有多少个 '1' 的连通域**

### ②并查集

```
int numIslands(vector<vector<char>>& grid) {
    UnionFind* uf = new UnionFind(grid.size()*grid[0].size());
    for(int i = 0; i < grid.size(); i++){
        for(int j = 0; j < grid[0].size(); j++){
            if(grid[i][j] == '1'){
                if(i > 0 && grid[i-1][j] == '1') {
                    union_nodes(uf, grid[0].size()*i+j, grid[0].size()*(i-1)+j);
                }
                if(j < grid.size()-1 && grid[i][j+1] == '1') {
                    union_nodes(uf, grid[0].size()*i+j, grid[0].size()*(i+1)+j);
                }
                if(j > 0 && grid[i][j-1] == '1') {
                    union_nodes(uf, grid[0].size()*i+j, grid[0].size()*(i-1)+j-1);
                }
                if(j < grid[0].size()-1 && grid[i][j+1] == '1') {
                    union_nodes(uf, grid[0].size()*i+j, grid[0].size()*(i+1)+j+1);
                }
            }
        }
    }
    int ent = 0;
    for(int i = 0; i < grid.size(); i++){
        for(int j = 0; j < grid[0].size(); j++){
            if((grid[i][j] == '1') && (find_parent(uf, grid[0].size()*i+j) == grid[0].size()*i+j)) {ent++;}
        }
    }
    return ent;
}
```

### 3. 新增变量 cnt、max-cnt, 累积BFS

```
for(int i = 0; i < grid.size(); i++){
    for(int j = 0; j < grid[0].size(); j++){
        if(grid[i][j] == 1){
            max_cnt = max(max_cnt, bfs(grid, i, j, visited, q_bfs, direction));
        }
    }
}
q_bfs.push(pos);
cnt++;
```

### 4. 新增变量 area、areas, 累积DFS

```

for(int i = 0; i<land.size(); i++){
    for(int j = 0; j<land[0].size(); j++){
        if(land[i][j] == 0 && !visited[i][j]){
            dfs(land, i, j, visited, direction, area);
            areas.emplace_back(area);
            area = 0;
        }
    }
}

```

dfs函数中一开始

```

visited[x][y] = true;
area++;

```

## 5.

你现在手里有一份大小为  $n \times n$  的网格 grid，上面的每个单元格都用 0 和 1 标记好了。其中 0 代表海洋，1 代表陆地。

请你找出一个海洋单元格，这个海洋单元格到离它最近的陆地单元格的距离是最大的，并返回该距离。如果网格上只有陆地或者海洋，请返回 -1。

我们这里说的距离是「曼哈顿距离」（Manhattan Distance）： $(x_0, y_0)$  和  $(x_1, y_1)$  这两个单元格之间的距离是  $|x_0 - x_1| + |y_0 - y_1|$ 。

示例 1：

1	0	1
0	0	0
1	0	1

输入：grid = [[1,0,1],[0,0,0],[1,0,1]]

输出：2

解释：

海洋单元格 (1, 1) 和所有陆地单元格之间的距离都达到最大，最大距离为 2。

## ① 多源BFS

```

int maxDistance(vector<vector<int>>& grid) {
    queue<vector<int>> q_bfs;
    vector<vector<int>> direction = {{1,0}, {-1,0}, {0,1}, {0,-1}};
    vector<vector<int>> dis(grid.size(), vector<int>(grid[0].size(), 0));
    int max_dis = 0;
    for(int i = 0; i<grid.size(); i++){
        for(int j = 0; j<grid[0].size(); j++){
            if(grid[i][j] == 1){ }  
首先将所有陆地推入队列
            vector<int> pos = {i,j};
            q_bfs.push(pos);
        }
    }
    if(q_bfs.size() == 0 || q_bfs.size() == grid.size()*grid[0].size()) {return -1;}
    bfs(grid, q_bfs, direction, dis, max_dis);
    return max_dis;
}

void bfs(queue<vector<int>>& q_bfs, vector<vector<int>>& grid, queue<vector<int>>& q_bfs, vector<vector<int>>& direction, vector<vector<int>>& dis, int& max_dis){
    while(!q_bfs.empty()){
        int x = q_bfs.front()[0];
        int y = q_bfs.front()[1];
        q_bfs.pop();
        for(int i = 0; i<4; i++){
            int new_x = x + direction[i][0];
            int new_y = y + direction[i][1];
            if(new_x<0 && new_x<grid.size() && new_y<0 && new_y<grid[0].size() && if(grid[new_x][new_y] == 1){continue};  
grid[new_x][new_y] = 1;
            vector<int> pos = {new_x, new_y};
            q_bfs.push(pos);
            dis[new_x][new_y] = dis[x][y] + 1;
            max_dis = max(max_dis, dis[new_x][new_y]);
        }
    }
}

```

## ③ Dijkstra

```

int maxDistance(vector<vector<int>>& grid) {
    vector<vector<int>> direction = {{1,0}, {-1,0}, {0,1}, {0,-1}};
    queue<vector<int>> q_dij;
    vector<vector<int>> dis(grid.size(), vector<int>(grid[0].size(), INT_MAX));
    int max_dis = 0;
    for(int i = 0; i<grid.size(); i++){
        for(int j = 0; j<grid[0].size(); j++){
            if(grid[i][j] == 1){
                vector<int> pos = {i, j};
                q_dij.push(pos);
                dis[i][j] = 0;
            }
        }
    }
    if(q_dij.size() == 0 || q_dij.size() == grid.size()*grid[0].size()){
        return -1;
    }
    dijkstra(grid, direction, q_dij, dis, max_dis);
    return max_dis;
}

void dijkstra(vector<vector<int>>& grid, vector<vector<int>>& direction, queue<vector<int>>& q_dij, vector<vector<int>>& dis,
    while(!q_dij.empty()){
        int x = q_dij.front()[0];
        int y = q_dij.front()[1];
        q_dij.pop();
        for(int i = 0; i<4; i++){
            int new_x = x + direction[i][0];
            int new_y = y + direction[i][1];
            if(new_x<=0 && new_x<grid.size() && new_y<=0 && new_y<grid[0].size()){
                if(dis[x][y] + 1 < dis[new_x][new_y]){
                    dis[new_x][new_y] = dis[x][y] + 1;
                    vector<int> pos = {new_x, new_y};
                    q_dij.push(pos);
                    max_dis = max(max_dis, dis[new_x][new_y]);
                }
            }
        }
    }
}

```

## 6.

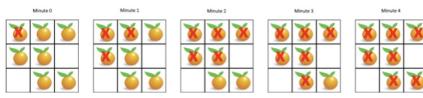
在给定的  $m \times n$  网格  $grid$  中，每个单元格可以有以下三个值之一：

- 值 0 表示空单元格；
- 值 1 表示新鲜橘子；
- 值 2 表示腐烂的橘子。

每分钟，腐烂的橘子 周围 4 个方向上相邻 的新鲜橘子都会腐烂。

返回 直到单元格中没有新鲜橘子为止所必须经过的最小分钟数。如果不可能，返回  $-1$ 。

示例 1：



输入:  $grid = [[2,1,1],[1,1,0],[0,1,1]]$   
输出: 4

和题5相似，时间实际上相当于距离

使用多源BFS

```

int orangesRotting(vector<vector<int>>& grid) {
    vector<vector<int>> direction = {{1,0}, {-1,0}, {0,1}, {0,-1}};
    queue<vector<int>> q_bfs;
    vector<vector<int>> dis(grid.size(), vector<int>(grid[0].size(), 0));
    int max_dis = 0;
    int fresh_cnt = 0;
    for(int i = 0; i<grid.size(); i++){
        for(int j = 0; j<grid[0].size(); j++){
            if(grid[i][j] == 1) {
                vector<int> pos = {i,j};
                q_bfs.push(pos);
            } else if(grid[i][j] == 2) {
                fresh_cnt++;
            }
        }
    }
    bfs(grid, direction, q_bfs, dis, max_dis, fresh_cnt);
    if(fresh_cnt == 0) {return -1;}
    else {return max_dis;}
}

```

```

void bfs(vector<vector<int>>& grid, vector<vector<int>> direction, queue<vector<int>>& q_bfs, vector<vector<int>>& dis, int& max_dis, int& max_dis, int& fresh_cnt){
    while(!q_bfs.empty()){
        int x = q_bfs.front()[0];
        int y = q_bfs.front()[1];
        q_bfs.pop();
        for(int i = 0; i<4; i++){
            int new_x = x + direction[i][0];
            int new_y = y + direction[i][1];
            if(new_x<=0 && new_x<grid.size() && new_y<=0 && new_y<grid[0].size()){
                if(grid[new_x][new_y] == 1){
                    dis[new_x][new_y] = dis[x][y] + 1;
                    q_bfs.push(pos);
                    grid[new_x][new_y] = 2;
                    max_dis = max(max_dis, dis[new_x][new_y]);
                    fresh_cnt--;
                    if(fresh_cnt == 0) {break;} // 可提前中断
                }
            }
        }
    }
}

```

## 拓扑排序的题目

不算BFS，只是思路类似。

1. 207. 课程表：你这个学期必须选修numCourses门课程，记为0到numCourses-1。在选修某些课程之前需要一些先修课程。例如，想要学习课程0，你需要先完成课程1，我们用一个匹配来表示他们：[0,1]。给定课程总量以及它们的先决条件，请你判断是否可能完成所有课程的学习？

2. 210. 课程表 II：现在你总共有n门课需要选，记为0到n-1。在选修某些课程之前需要一些先修课程。例如，想要学习课程0，你需要先完成课程1，我们用一个匹配来表示他们：[0,1]。给定课程总量以及它们的先决条件，返回你为了学完所有课程所安排的学习顺序。可能会有多个正确的顺序，你只要返回一种就可以了。如果不可能完成所有课程，返回一个空数组。

提升：本地编译环境试试返回所有正确的顺序。（返回值类型改变）

3. 630. 课程表 III：这里有n门不同的在线课程，他们按从1到n编号。每一门课程有一定的持续上课时间（课程时间）t以及关闭时间第d天。一门课要持续学习t天直到第d天时要完成，你将会从第1天开始。给出n个在线课程用(t, d)对表示。你的任务是找出最多可以修几门课。

此题可以用拓扑排序，但是可能贪心更方便，题目较难，放在这就是为了系列题好看而已，看不懂的建议直接跳过。

- 这道题使用的算法有一个专有的名词，叫「拓扑排序」；
- 但是本质是依然是应用了经典的算法思想：「广度优先遍历」、「贪心算法」；
- 「拓扑排序」是「广度优先遍历」和「贪心算法」应用于「有向图」的一个专有名词；
- 应用场景：任务调度计划、课程安排，例：《机器学习》课程的先导课程为《高等数学》；
- 「拓扑排序」的作用：
  - 得到一个「拓扑序」，「拓扑序」不唯一；
  - 检测「有向图」是否有环。
- 补充：「无向图」中检测是否有环，使用的数据结构是「并查集」。

拓扑排序实际上应用的是贪心算法。贪心算法简而言之：每一步最优，全局就最优。

具体到拓扑排序，每一次都从图中删除没有前驱的顶点，这里并不需要真正的做删除操作，我们可以设置一个入度数组，每一轮都输出入度为0的结点，并移除它，修改它指向的结点的入度（-1即可），依次得到的结点序列就是拓扑排序的结点序列。如果图中还有结点没有被移除，则说明“不能完成所有课程的学习”。

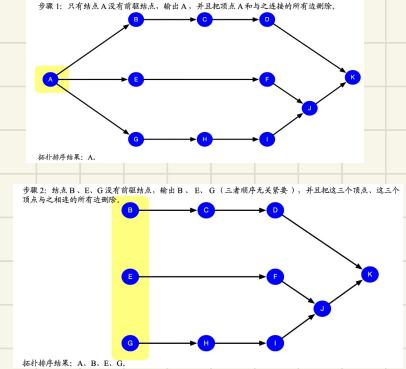
拓扑排序保证了每个活动（在这题中是“课程”）的所有前驱活动都排在该活动的前面，并且可以完成所有活动。拓扑排序的结果不唯一。拓扑排序还可以用于检测一个有向图是否有环。相关的概念还有AOV网，这里就不展开了。

在代码具体实现的时候，除了保存入度为0的队列，我们还需要两个辅助的数据结构：

- 邻接表：通过结点的索引，我们能够得到这个结点的后继结点；`vector<vector<int>> edges`
- 入度数组：通过结点的索引，我们能够得到指向这个结点的结点个数。`vector<int> indeg`

这两个数据结构在遍历题目给出的邻边以后就可以很方便地得到。

```
bool canFinish(int numCourses, vector<vector<int>>& prerequisites) {  
    vector<int> indeg(numCourses);  
    vector<vector<int>> edges(numCourses);  
    queue<int> q_top;  
    int cnt = 0;  
    for(auto& p:prerequisites){  
        indeg[p[0]]++;  
        edges[p[1]].emplace_back(p[0]);  
    }  
    for(int i = 0; i<numCourses; i++){  
        if(indeg[i] == 0) {q_top.push(i);} 把入度为0的结点入队列  
    }  
    while(!q_top.empty()){  
        int top = q_top.front();  
        q_top.pop();  
        cnt++;  
        for(auto& v:edges[top]){  
            indeg[v]--;  
            if(indeg[v] == 0) {q_top.push(v);} 入度为0则入队列  
        }  
    }  
    return cnt == numCourses;  
}
```



## 2. 相比题1墙表了

```
q_top.pop();
cnt++;
ans.emplace_back(top);
```

```
if(cnt != numCourses) {ans.resize(0);}
return ans;
```

## 3. 要使用贪心算法

### 染色的题目

不算BFS，只是思路类似。

- 733. 图像渲染**: 有一幅以二维整数数组表示的图画，每一个整数表示该图画的像素值大小，数值在 0 到 65535 之间。给你一个坐标 (sr, sc) 表示图像渲染开始的像素值（行，列）和一个新的颜色值 newColor，让你重新上色这幅图像。
- 面试题 08.10. 颜色填充**: 题目名称和题目叙述与上一题不太一样，代码可以一模一样。
- 1042. 不邻接植花**: 有 n 个花园，按从 1 到 n 标记。另有数组 paths，其中 paths[i] = [xi, yi] 描述了花园 xi 到花园 yi 的双向路径。在每个花园中，你打算种下四种花之一。另外，所有花园最多有3条路径可以进入或离开。你需要为每个花园选择一种花，使得通过路径相连的任何两个花园中的花的种类互不相同。

本质上也是个染色问题。

1.

为了完成“上色工作”，从初始像素开始，记录初始坐标上的上下左右四个方向上像素值与初始坐标相同的相连像素点，接着再记录这四个方向上符合条件的像素点与他们对应四个方向上像素值与初始坐标相同的相连像素点，……，重复该过程，将所有记录的像素点的颜色值改为 newColor。

最后返回经过上色渲染后的图像。

示例 1：

1	1	1
1	1	0
1	0	1

即“油漆桶”功能  
使用BFS

## 2. 与题1相同

3. 以数组形式返回任一可行的方案作为答案 answer，其中 answer[i] 为在第 (i+1) 个花园中种植的花的种类。花的种类用 1、2、3、4 表示。保证存在答案。

```
vector<int> gardenNoAdj(int n, vector<vector<int>>& paths) {
    vector<vector<int>> garden_map(n);
    vector<int> colors(n, 0);
    for(int i = 0; i < paths.size(); i++){
        garden_map[paths[i][0]-1].emplace_back(paths[i][1]-1);
        garden_map[paths[i][1]-1].emplace_back(paths[i][0]-1);
    }
    for(int i = 0; i < n; i++){
        if(colors[i] == 0) {bfs_change(i, garden_map, colors);}
    }
    return colors;
}

void bfs_change(int i, vector<vector<int>>& garden_map, vector<int>& colors){
    vector<bool> colors_visited(4, false);
    for(auto& g:garden_map[i]){
        int g_color = colors[g];
        colors_visited[g_color] = true;
    }
    for(int c = 1; c <= 4; c++){
        if(!colors_visited[c]){
            colors[i] = c;
            break;
        }
    }
}
```

# 題399除法求值 (難題)

BFS

```
vector<double> calcEquation(vector<vector<string>>& equations, vector<double>& values, vector<vector<string>>& queries) {
    int nvars = 0;
    unordered_map<string, int> variables;

    int n = equations.size();
    for (int i = 0; i < n; i++) {
        if (variables.find(equations[i][0]) == variables.end()) {
            variables[equations[i][0]] = nvars++;
        }
        if (variables.find(equations[i][1]) == variables.end()) {
            variables[equations[i][1]] = nvars++;
        }
    }

    // 对于每个点，存储其直接连接到的所有点及对应的权值
    vector<vector<pair<int, double>>> edges(nvars);
    for (int i = 0; i < n; i++) {
        int va = variables[equations[i][0]], vb = variables[equations[i][1]];
        edges[va].push_back(make_pair(vb, values[i]));
        edges[vb].push_back(make_pair(va, 1.0 / values[i]));
    }

    vector<double> ret;
    for (const auto& q: queries) {
        double result = -1.0;
        if (variables.find(q[0]) != variables.end() && variables.find(q[1]) != variables.end()) {
            int ia = variables[q[0]], ib = variables[q[1]];
            if (ia == ib) {
                result = 1.0;
            } else {
                queue<int> points;
                points.push(ia);
                vector<double> ratios(nvars, -1.0);
                ratios[ia] = 1.0;

                while (!points.empty() && ratios[ib] < 0) {
                    int x = points.front();
                    points.pop();

                    for (const auto [y, val]: edges[x]) {
                        if (ratios[y] < 0) {
                            ratios[y] = ratios[x] * val;
                            points.push(y);
                        }
                    }
                }
                result = ratios[ib];
            }
        }
        ret.push_back(result);
    }
    return ret;
}
```

# 拓展：310. 最小高度树

树是一个无向图，其中任何两个顶点只通过一条路径连接。换句话说，一个任何没有简单环路的连通图都是一棵树。

给你一棵包含  $n$  个节点的树，标记为 0 到  $n - 1$ 。给定数字  $n$  和一个有  $n - 1$  条无向边的 edges 列表（每一个边都是一对标签），其中  $\text{edges}[i] = [a_i, b_i]$  表示树中节点  $a_i$  和  $b_i$  之间存在一条无向边。

可选择树中任何一个节点作为根。当选择节点  $x$  作为根节点时，设结果树的高度为  $h$ 。在所有可能的树中，具有最小高度的树（即， $\min(h)$ ）被称为 **最小高度树**。

请你找到所有的 **最小高度树** 并按 **任意顺序** 返回它们的根节点标签列表。

树的 **高度** 是指根节点和叶子节点之间最长向下路径上边的数量。

## 方法1 → BFS (超时了)

```
vector<int> findMinHeightTrees(int n, vector<vector<int>>& edges) {
    vector<vector<int>> edges_map(n);
    vector<int> ans;
    int minheight = INT_MAX, height = 0;
    for(auto& e:edges) {
        edges_map[e[0]].emplace_back(e[1]);
        edges_map[e[1]].emplace_back(e[0]);
    }
    for(int i = 0; i<n; i++) {
        bfs(i, edges_map, height, n);
        if(height < minheight) {
            minheight = height;
            ans.resize(0);
            ans.emplace_back(i);
        }
        else if(height == minheight) {
            ans.emplace_back(i);
        }
        height = 0;
    }
    return ans;
}
void bfs(int i, vector<vector<int>>& edges_map, int& height, int n) {
    queue<int> q;
    vector<bool> visited(n, false);
    q.push(i);
    while(!q.empty()) {
        int size = q.size();
        for(int m = 0; m<size; m++) {
            int tmp = q.front();
            q.pop();
            visited[tmp] = true;
            for(auto& e:edges_map[tmp]) {
                if(!visited[e]) {q.push(e);}
            }
        }
        height++;
    }
}
```

## 方法2 → 拓扑排序 相当于从叶结点到根节点的拓扑

由于树的高度由根节点到叶子节点之间的最大距离构成，假设树中距离最长的两个节点为  $(x, y)$ ，它们之间的距离为  $\max\text{dist} = \text{dist}[x][y]$ ，假设  $x$  到  $y$  的路径为  $x \rightarrow p_1 \rightarrow p_2 \rightarrow \dots \rightarrow p_{k-1} \rightarrow p_k \rightarrow y$ ，根据方法一的证明已知最小树的根节点一定为该路径中的中间节点，我们尝试删除最外层的度为 1 的节点  $x, y$  后，则可以知道路径中与  $x, y$  相邻的节点  $p_1, p_k$  此时也变为度为 1 的节点，此时我们再次删除最外层度为 1 的节点直到剩下根节点为止。

```
vector<int> findMinHeightTrees(int n, vector<vector<int>>& edges) {
    vector<vector<int>> edges_map(n);
    vector<int> degree(n, 0);
    queue<int> q;
    vector<int> ans;
    if(n == 1) {ans.emplace_back(0); return ans;}
    for(auto& e:edges) {
        edges_map[e[0]].emplace_back(e[1]);
        edges_map[e[1]].emplace_back(e[0]);
        degree[e[0]]++; degree[e[1]]++;
    }
    for(int i = 0; i<n; i++) {
        if(degree[i] == 1) {q.push(i);}
    }
    while(!q.empty()) {
        int size = q.size();
        ans.resize(0);
        for(int m = 0; m<size; m++) {
            int tmp = q.front();
            q.pop();
            degree[tmp]--;
            ans.emplace_back(tmp);
            for(auto& e:edges_map[tmp]) {
                degree[e]--;
                if(degree[e] == 1) {q.push(e);}
            }
        }
    }
    return ans;
}
```

# 4. 链表

## 基本操作

```
/*创建链表*/
class ListNode{
public:
    int val;
    ListNode* next;
    ListNode() {this->val = 0; this->next = nullptr;};
    ListNode(int val) {this->val = val; this->next = nullptr;};
    ListNode(int val, ListNode* next) {this->val = val; this->next = next;};
};
```

```
/*生成链表*/
ListNode* createList(vector<int>& nums){
    ListNode* pre = new ListNode();
    ListNode* cur = pre;
    for(auto& num:nums){
        cur->next = new ListNode(num);
        cur = cur->next;
    }
    return pre->next;
}
```

```
/*输出链表*/
void printList(ListNode* head){
    while(head != nullptr){
        if(head->next != nullptr) {cout<<head->val<<"->"};
        else {cout<<head->val;};
        head = head->next;
    }
}
```

## 迭代版

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



## 递归版

```
ListNode* reverseList(ListNode* head) {
    if(head == nullptr || head->next == nullptr) {return head;};
    ListNode* ret = reverseList(head->next);
    head->next->next = head;
    head->next = nullptr;
    return ret;
}
```



## 链表中的加法题

**提示：**不要因为简单而轻视这类题目，多思考边界条件及其变型！

1. 2. **两数相加：**反向存放、反向输出。
2. **面试题 02.05. 链表求和：**与上一题相比，有更多的边界条件。
3. **445. 两数相加 II：**正向存放，正向输出。推荐看[@sweetiee的简单Java](#)

# 1. 2. 3.

给定两个用链表表示的整数，每个节点包含一个数位。

这些数位是反向存放的，也就是个位排在链表首部。

编写函数对这两个整数求和，并用链表形式返回结果。

示例：

输入：(7 -> 1 -> 6) + (5 -> 9 -> 2)，即  $617 + 295$   
输出：2 -> 1 -> 9，即  $912$

进阶：思考一下，假设这些数位是正向存放的，又该如何解决呢？

示例：

输入：(6 -> 1 -> 7) + (2 -> 9 -> 5)，即  $617 + 295$   
输出：9 -> 1 -> 2，即  $912$

双向：

```
ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
    ListNode* pre = new ListNode(); → 使用一个空头节点
    ListNode* cur = pre;
    int sum = 0;
    bool plus = false;
    while(l1 != nullptr && l2 != nullptr){
        if(plus) {sum = l1->val + l2->val + 1;}
        else {sum = l1->val + l2->val;}
        if(sum/10 == 1) {plus = true;}
        else {plus = false;}
        cur->next = new ListNode(sum%10);
        cur = cur->next;
        l1 = l1->next;
        l2 = l2->next;
    }
    while(l1 != nullptr){
        if(plus) {sum = l1->val + 1;}
        else {sum = l1->val;}
        if(sum/10 == 1) {plus = true;}
        else {plus = false;}
        cur->next = new ListNode(sum%10);
        cur = cur->next;
        l1 = l1->next;
    }
    while(l2 != nullptr){
        if(plus) {sum = l2->val + 1;}
        else {sum = l2->val;}
        if(sum/10 == 1) {plus = true;}
        else {plus = false;}
        cur->next = new ListNode(sum%10);
        cur = cur->next;
        l2 = l2->next;
    }
    if(plus) {cur->next = new ListNode(1); cur = cur->next;}
    return pre->next;
}
```

正向：

```
ListNode* addTwoNumbers(ListNode* l1, ListNode* l2) {
    stack<int> s1, s2;
    while(l1 != NULL){
        s1.push(l1->val);
        l1 = l1->next;
    }
    while(l2 != NULL){
        s2.push(l2->val);
        l2 = l2->next;
    }
    ListNode* pre = NULL; → 使用一个空指针节点
    int sum = 0;
    bool plus = false;
    while(!s1.empty() && !s2.empty()){
        if(plus) {sum = s1.top() + s2.top() + 1;}
        else {sum = s1.top() + s2.top();}
        s1.pop(); s2.pop();
        if(sum/10 == 1) {plus = true;}
        else {plus = false;}
        ListNode* cur = new ListNode(sum%10);
        cur->next = pre;
        pre = cur;
    }
    while(!s1.empty()){
        if(plus) {sum = s1.top() + 1;}
        else {sum = s1.top();}
        s1.pop();
        if(sum/10 == 1) {plus = true;}
        else {plus = false;}
        ListNode* cur = new ListNode(sum%10);
        cur->next = pre;
        pre = cur;
    }
    while(!s2.empty()){
        if(plus) {sum = s2.top() + 1;}
        else {sum = s2.top();}
        s2.pop();
        if(sum/10 == 1) {plus = true;}
        else {plus = false;}
        ListNode* cur = new ListNode(sum%10);
        cur->next = pre;
        pre = cur;
    }
    if(plus) {ListNode* cur = new ListNode(1); cur->next = pre; pre = cur;}
    return pre;
}
```

## 反转链表的题

注明：以下某些题目是可以用（但不限于用）反转链表思路的题目。

1. 206. 反转链表 和 动指 Offer 24. 反转链表：一模一样的基础题，建议不懂反转链表套路的同学翻一翻这两题的题解区，找到对应语言的大佬们看看他们对比如的解释（有画图很棒的大佬）。

2. 92. 反转链表 II：反转从位置m到n的链表。请使用一趟扫描完成反转。

3. 动指 Offer 06. 从尾到头打印链表：输入一个链表的头节点，从尾到头反过来返回每个节点的值（用数组返回）。

4. 143. 重排链表：不能只是单纯的改变节点内部的值，而是需要实际的进行节点交换。

5. 25. K 个一组翻转链表：每k个节点一组进行翻转，返回翻转后的链表。PS：困难题看不懂的时候及时放弃。

6. 234. 回文链表 和 面试题 02.06. 回文链表：判断一个链表是否为回文链表。

# 1. 基本操作

## 2. 两次遍历:

- 第1步：先将待反转的区域反转；
- 第2步：把 pre 的 next 指针指向反转以后的链表头节点，把反转以后的链表的尾节点的 next 指针指向 succ。



### 一次遍历:

整体思想是：在需要反转的区间里，每遍历到一个节点，让这个新节点来到反转部分的起始位置。下面的图展示了整个流程。



#### 操作步骤：

- 先将 curr 的下一个节点记录为 next；
- 执行操作 ①：把 curr 的下一个节点指向 next 的下一个节点；
- 执行操作 ②：把 next 的下一个节点指向 pre 的下一个节点；
- 执行操作 ③：把 pre 的下一个节点指向 next。

```

ListNode* reverseBetween(ListNode* head, int left, int right) {
    ListNode* dummy = new ListNode(); // 因为头节点有可能发生变化，使用虚拟头节点可以避免复杂的分类讨论
    dummy->next = head;
    ListNode* pre = dummy;
    for(int i = 0; i<left-1; i++) {pre = pre->next;}
    ListNode* leftnode = pre->next;
    ListNode* rightnode = pre;
    for(int i = 0; i<right-left+1; i++) {rightnode = rightnode->next;}
    ListNode* suc = rightnode->next;
    // 断开链接
    pre->next = nullptr;
    rightnode->next = nullptr;
    // 反转链表的子区间
    reverseList(leftnode); => 基本操作的简化版
    // 恢复原来的链表中
    pre->next = rightnode;
    leftnode->next = suc;
    return dummy->next;
}
  
```

```

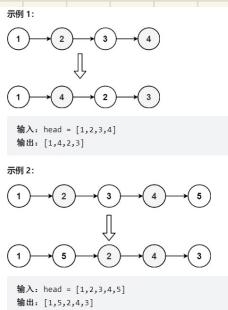
ListNode* reverseBetween(ListNode* head, int left, int right) {
    基本操作步骤 使用 dummy
    ListNode* dummy = new ListNode();
    dummy->next = head;
    ListNode* pre = dummy;
    for(int i = 0; i<left-1; i++) {pre = pre->next;}
    ListNode* cur = pre->next;
    ListNode* next;
    for(int i = 0; i<right-left; i++){
        next = cur->next;
        cur->next = next->next;
        next->next = pre->next;
        pre->next = next;
    }
    return dummy->next;
}
  
```

## 3.

输入: head = [1,3,2]  
输出: [2,3,1]

**使用辅助栈即可**

## 4.



### 寻找链表中点 + 链表反转 + 链表合并

这样我们的任务即可划分为三步：

- 找到原链表的中点（参考「[876. 链表的中间结点](#)」）。
  - 我们可以使用快慢指针实现  $O(N)$  地找到链表的中间节点。
- 将原链表的右半端反转（参考「[206. 反转链表](#)」）。
  - 我们可以使用迭代法实现链表的反转。
- 将原链表的两端合并。
  - 因为两链表长度相差不超过 1，因此直接合并即可。

```

void reorderList(ListNode* head) {
    ① ListNode* middleNode = middleList(head);
    ListNode* L1 = head;
    ListNode* L2 = middleNode;
    ② L2 = reverseList(L2);
    ③ mergeList(L1, L2);
    middleNode->next = nullptr; 最后一个结点的next为nullptr
}

```

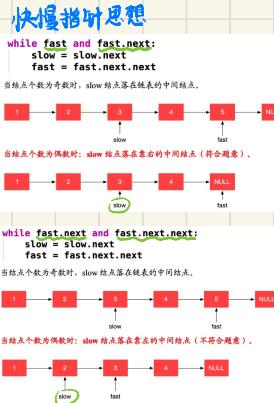
```

void mergeList(ListNode* L1, ListNode* L2){
    ListNode* tmp1;
    ListNode* tmp2;
    while(L1 != nullptr && L2 != nullptr){
        tmp1 = L1->next;
        tmp2 = L2->next;

        L1->next = L2;
        L1 = tmp1;

        L2->next = L1;
        L2 = tmp2;
    }
}

```

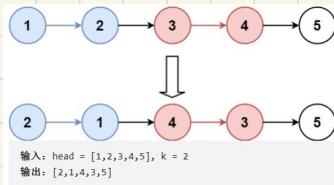


```

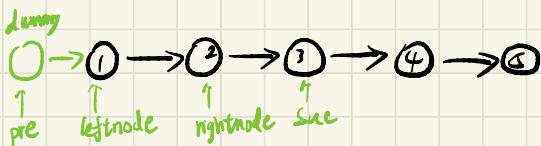
ListNode* middleList(ListNode* head){
    ListNode* slow = head;
    ListNode* fast = head;
    while(fast != nullptr && fast->next != nullptr){
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}

```

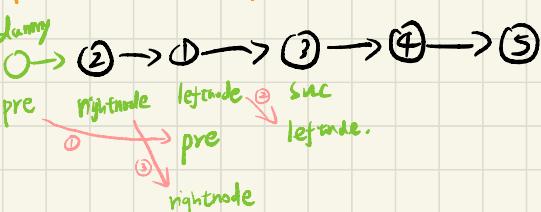
## 5. 题2两次遍历的提升



第1组反转后



第2组反转前指针更新

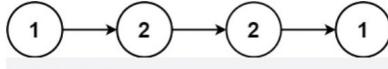


```

ListNode* reverseKGroup(ListNode* head, int k) {
    ListNode* dummy = new ListNode();
    dummy->next = head;
    ListNode* pre = dummy;
    ListNode* leftnode = pre->next;
    ListNode* rightnode = pre;
    while(1){
        for(int i = 0; i < k; i++){
            rightnode = rightnode->next;
            if(rightnode == nullptr) {return dummy->next;}
        }
        ListNode* suc = rightnode->next;
        pre->next = nullptr; | 切断连接
        rightnode->next = nullptr; | 反转子链表
        reverseList(leftnode); | 反转子链表
        pre->next = rightnode; | 拼接到原处链表
        leftnode->next = suc; | 拼接
        pre = leftnode; | 指针更新
        leftnode = suc; | 指针更新
        rightnode = pre;
    }
    return dummy->next;
}

```

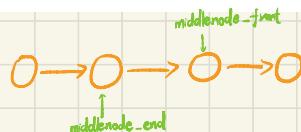
6.



输入: head = [1,2,2,1]  
输出: true

1. 找到链表中点
2. 反转后半部分
3. 前半部分与后半部分一一比较

```
bool isPalindrome(ListNode* head) {
    ListNode* middleNode_end = middleList(head); // 得出第一组的中点
    ListNode* middleNode_front = reverseList(middleNode_end->next);
    middleNode_end->next = middleNode_front;
    while(middleNode_front != nullptr){
        if(head->val != middleNode_front->val) {return false;}
        head = head->next;
        middleNode_front = middleNode_front->next;
    }
    return true;
}
```



## 相交链表的题

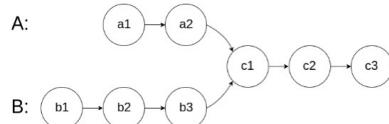
**注明:** 这三题可以用同一个代码，大家可以自行揣摩题目的区别，反正代码是没啥区别。

1. 160. 相交链表
2. 力扣 Offer 52. 两个链表的第一个公共节点
3. 面试题 02.07. 链表相交

1.2.3.

给你两个单链表的头节点 headA 和 headB，请你找出并返回两个单链表相交的起始节点。如果两个链表不存在相交节点，返回 null。

图示两个链表在节点 c1 开始相交：



### ① 使用哈希表辅助记录

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    set list_map;
    ListNode* tmp = headA;
    while(tmp != NULL) {
        list_map.insert(tmp);
        tmp = tmp->next;
    }
    tmp = headB;
    while(tmp != NULL) {
        if(list_map.count(tmp)) {return tmp;}
        tmp = tmp->next;
    }
    return NULL;
}
```

### ② 双指针

```
ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
    ListNode* tmp1 = headA;
    ListNode* tmp2 = headB;
    while(tmp1 != tmp2) {
        if(tmp1 == NULL) {tmp1 = headB;}
        else {tmp1 = tmp1->next;}
        if(tmp2 == NULL) {tmp2 = headA;}
        else {tmp2 = tmp2->next;}
    }
    return tmp1;
}
```

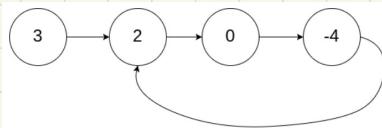
由于若链表有相交部分，则两个指针 tmp1, tmp2 在分别遍历 headA, headB 过程中必会在某一时刻在相交节点相遇。

<https://leetcode.cn/problems/intersection-of-two-linked-lists/solution/intersection-of-two-linked-lists-shuang-zhi-zhen-1/>

# 环形链表

1. 141. 环形链表：判断有没有环
2. 142. 环形链表 II：返回链表开始入环的第一个节点
3. 面试题 02.08. 环路检测：跟上题一样

1.



输入: head = [3,2,0,-4], pos = 1  
输出: true  
解释: 链表中有一个环，尾部连接到第二个节点。

## ① 使用哈希表辅助记录

```
bool hasCycle(ListNode *head) {
    set<ListNode*> list_map;
    while(head != NULL){
        if(list_map.count(head)) {return true;}
        list_map.insert(head);
        head = head->next;
    }
    return false;
}
```

2.3.

## ① 使用哈希表辅助记录：很容易实现

## ② 快慢指针

我们使用两个指针,  $fast$  与  $slow$ , 它们起始都位于链表的头部。随后,  $slow$  指针每次向后移动一个位置, 而  $fast$  指针向后移动两个位置。如果链表中存在环, 则  $fast$  指针最终将再次与  $slow$  指针在环中相遇。

如下图所示, 设链表中环外部分的长度为  $a$ 。 $slow$  指针进入环后, 又走了  $b$  的距离与  $fast$  相遇。此时,  $fast$  指针已经走完了环的  $n$  圈, 因此它走过的总距离为  $a + n(b + c) + b = a + (n + 1)b + nc$ 。

根据题意, 任意时刻,  $fast$  指针走过的距离都为  $slow$  指针的 2 倍。因此, 我们有

$$a + (n + 1)b + nc = 2(a + b) \implies a = c + (n - 1)(b + c)$$

有了  $a = c + (n - 1)(b + c)$  的等量关系, 我们会发现: 从相遇点到入环点的距离加上  $n - 1$  圈的环长, 恰好等于从链表头部到入环点的距离。

因此, 当发现  $slow$  与  $fast$  相遇时, 我们再额外使用一个指针  $ptr$ 。起始, 它指向链表头部; 随后, 它和  $slow$  每次向后移动一个位置。最终, 它们会在入环点相遇。

## ② 快慢指针

假想「乌龟」和「兔子」在链表上移动, 「兔子」跑得快, 「乌龟」跑得慢。当「乌龟」和「兔子」从链表上的同一个节点开始移动时, 如果该链表中没有环, 那么「兔子」将一直处于「乌龟」的前方; 如果该链表中有环, 那么「兔子」会先于「乌龟」进入环, 并且一直在环内移动。等到「乌龟」进入环时, 由于「兔子」的速度快, 它一定会在某个时刻与乌龟相遇, 即套了「乌龟」若干圈。

```
bool hasCycle(ListNode *head) {
    if(head == NULL || head->next == NULL) {return false;}
    ListNode* slow = head;
    ListNode* fast = head->next;
    while(slow != fast){
        if(fast == NULL || fast->next == NULL) {return false;}
        slow = slow->next;
        fast = fast->next->next;
    }
    return true;
}
```

```
ListNode *detectCycle(ListNode *head) {
    if(head == NULL || head->next == NULL) {return NULL;}
    ListNode* slow = head;
    ListNode* fast = head;
    while(fast != NULL){
        if(fast->next == NULL) {return NULL;}
        slow = slow->next;
        fast = fast->next->next;
        if(slow == fast){
            ListNode* ans = head;
            while(ans != slow){
                ans = ans->next;
                slow = slow->next;
            }
            return ans;
        }
    }
    return NULL;
}
```

## 倒数第k个节点

**注明：**都是用快慢指针法

1. 剑指 Offer 22. 链表中倒数第k个节点：返回的是节点
2. 面试题 02.02. 返回倒数第 k 个节点：返回的是节点的值 (val)

1. 初始化：前指针 former、后指针 latter，双指针都指向头节点 head。
2. 构建双指针距离：前指针 former 先向前走 k 步（结束后，双指针 former 和 latter 间相距 k 步）。
3. 双指针共同移动：循环中，双指针 former 和 latter 每轮都向前走一步，直至 former 走过链表尾节点时跳出（跳出后，latter 与尾节点距离为 k - 1，即 latter 指向倒数第 k 个节点）。
4. 返回值：返回 latter 即可。

```
ListNode* getKthFromEnd(ListNode* head, int k) {
    ListNode* slow = head;
    ListNode* fast = head;
    for(int i = 0; i < k; i++) {
        fast = fast->next;
    }
    while(fast != NULL){
        slow = slow->next;
        fast = fast->next;
    }
    return slow;
}
```

## 删除链表中的某个节点

**注明：**注意题目之间的差异和边界条件。好几题的题目长的差不多，点进去以后略有不同。

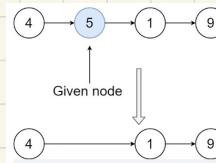
1. 剑指 Offer 18. 删除链表的节点
2. 237. 删除链表中的节点 和面试题 02.03. 删除中间节点
3. 19. 删除链表的倒数第N个节点
4. 83. 删除排序链表中的重复元素：给定一个排序链表，删除所有重复的元素，使得每个元素只出现一次。
5. 82. 删除排序链表中的重复元素 II：给定一个排序链表，删除所有含有重复数字的节点，只保留原始链表中没有重复出现的数字。
6. 面试题 02.01. 移除重复节点：移除未排序链表中的重复节点。保留最开始出现的节点。
7. 203. 移除链表元素：删除链表中等于给定值 val 的所有节点。

1.

输入: head = [4,5,1,9], val = 5  
输出: [4,1,9]  
解释: 给定你链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9。

```
ListNode* deleteNode(ListNode* head, int val) {
    ListNode* dummy = new ListNode();
    dummy->next = head;
    ListNode* pre = dummy, *cur = head;
    while(cur->val != val){
        cur = cur->next;
        pre = pre->next;
    }
    pre->next = cur->next;
    return dummy->next;
}
```

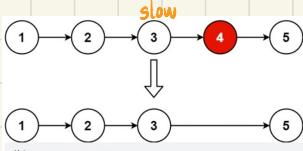
2. 相比题1，这题只输入要被删除节点。



输入: head = [4,5,1,9], node = 5  
输出: [4,1,9]  
解释: 指定链表中值为 5 的第二个节点，那么在调用了你的函数之后，该链表应变为 4 -> 1 -> 9。

```
void deleteNode(ListNode* node) {
    node->val = node->next->val;
    node->next = node->next->next;
}
```

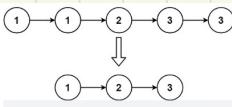
3.



输入: head = [1,2,3,4,5], n = 2  
输出: [1,2,3,5]

```
ListNode* removeNthFromEnd(ListNode* head, int n) {
    ListNode* dummy = new ListNode();
    dummy->next = head;
    ListNode* slow = dummy;
    ListNode* fast = head;
    for(int i = 0; i < n; i++){
        fast = fast->next;
    }
    while(fast != nullptr){
        slow = slow->next;
        fast = fast->next;
    }
    slow->next = slow->next->next;
    return dummy->next;
}
```

4.



输入: head = [1,1,2,3,3]  
输出: [1,2,3]

```
ListNode* deleteDuplicates(ListNode* head) {
    if(head == nullptr) {return head;}
    ListNode* dummy = new ListNode();
    dummy->next = head;
    ListNode* pre = head, *cur = head->next;
    while(cur != nullptr){
        if(pre->val != cur->val) {
            pre->next = cur;
            pre = cur;
        }
        cur = cur->next;
    }
    pre->next = nullptr;
    return dummy->next;
}
```

6.

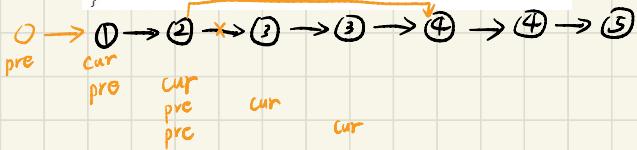
编写代码，移除未排序链表中的重复节点。保留最开始出现的节点。

示例1:

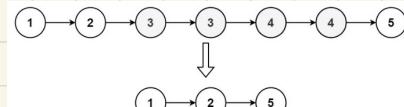
输入: [1, 2, 3, 3, 2, 1]  
输出: [1, 2, 3]

## ① 使用哈希表

```
ListNode* removeDuplicateNodes(ListNode* head) {
    set<int> list_map;
    ListNode* dummy = new ListNode();
    dummy->next = head;
    ListNode* pre = dummy, *cur = head;
    while(cur != NULL){
        if(list_map.count(cur->val)) {
            pre->next = cur->next;
        }
        else{
            list_map.insert(cur->val);
            pre = pre->next;
        }
        cur = cur->next;
    }
    return dummy->next;
}
```



5.



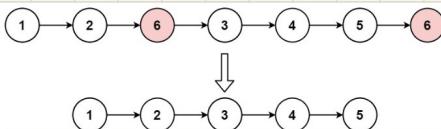
输入: head = [1,2,3,3,4,4,5]  
输出: [1,2,5]

```
ListNode* deleteDuplicates(ListNode* head) {
    if(head == nullptr) {return head;}
    ListNode* dummy = new ListNode();
    dummy->next = head;
    ListNode* pre = dummy, *cur = head;
    while(cur != nullptr){
        while(cur->next != nullptr && cur->next->val == cur->val){
            cur = cur->next;
        }
        if(pre->next == cur){  
            //说明 cur 没有在上面while循环放过, 因此没有重复  
            pre = pre->next;
        }
        else{
            pre->next = cur->next;
        }
        cur = cur->next;
    }
    return dummy->next;
}
```

## ② 不使用辅助空间

```
ListNode* removeDuplicateNodes(ListNode* head) {
    set<int> list_map;
    ListNode* dummy = new ListNode();
    dummy->next = head;
    ListNode* cur = head, *next = head;
    while(cur != NULL){
        while(next->next !=NULL){
            if(next->next->val == cur->val){
                next->next = next->next->next;
            }
            else{
                next = next->next;
            }
        }
        cur = cur->next;
        next = cur;
    }
    return dummy->next;
}
```

7.



```
ListNode* removeElements(ListNode* head, int val) {
    ListNode* dummy = new ListNode();
    dummy->next = head;
    ListNode* pre = dummy, *cur= head;
    while(cur != nullptr){
        if(cur->val == val){
            pre->next = cur->next;
        }
        else {
            pre = pre->next;
        }
        cur = cur->next;
    }
    return dummy->next;
}
```

## 排序链表

1. 147. 对链表进行插入排序：对链表进行插入排序。
2. 148. 排序链表：给你链表的头结点 head，请将其按升序排列并返回排序后的链表。**提升：**如何输出按降序排列的链表。

1. 详见排序章节，有插入排序、选择排序两种方法

## 2. 插入排序超时

### ① 归并排序

```
ListNode* sortList(ListNode* head) {  
    if(head == nullptr || head->next == nullptr) {return head;}  
    ListNode* slow = head;  
    ListNode* fast = head;  
    while(fast->next != nullptr && fast->next->next != nullptr){  
        slow = slow->next;  
        fast = fast->next->next;  
    }  
  
    ListNode* R_list = slow->next;  
    slow->next = nullptr;  
    ListNode* left = sortList(head);  
    ListNode* right = sortList(R_list);  
    ListNode* tmp = new ListNode();  
    ListNode* dummy = tmp;  
    while(left != nullptr && right != nullptr){  
        if(left->val < right->val){  
            tmp->next = left;  
            left = left->next;  
        }  
        else{  
            tmp->next = right;  
            right = right->next;  
        }  
        tmp = tmp->next;  
    }  
    if(left != nullptr){  
        tmp->next = left;  
    }  
    if(right != nullptr){  
        tmp->next = right;  
    }  
  
    return dummy->next;  
}
```

### ② 快速排序

```
ListNode* sortList(ListNode* head) {  
    if(head == nullptr) {return head;}  
    int min_val = head->val, max_val = head->val;  
    ListNode* cur = head, *suc = nullptr, *left = nullptr, *right = nullptr;  
    while(cur != nullptr){  
        min_val = min(min_val, cur->val);  
        max_val = max(max_val, cur->val);  
        cur = cur->next;  
    }  
    if(min_val == max_val) {return head;}  
    double pivot = (min_val + max_val)/2.0;  
    cur = head; ?  
    while(cur != nullptr){  
        suc = cur->next;  
        if(cur->val <= pivot){  
            cur->next = left;  
            left = cur;  
        }  
        else{  
            cur->next = right;  
            right = cur;  
        }  
        cur = suc;  
    }  
    left = sortList(left); //对小于pivot部分排序  
    right = sortList(right); //对大于pivot部分排序  
    cur = left;  
    while(cur->next != nullptr) {cur = cur->next;}  
    cur->next = right;  
    return left;  
}
```

## 合并链表

**注明：**第一题必会，后两题不会就算了吧，别纠结了。

1. **剑指 Offer 25. 合并两个排序的链表** 和 **21. 合并两个有序链表**：输入两个递增排序的链表，合并这两个链表并使新链表中的节点仍然是递增排序的。
2. **1669. 合并两个链表**：数字比较大的多半是竞赛题
3. **23. 合并K个升序链表**：困难题，量力而行

# 1. 类似归并排序的后半部分

```

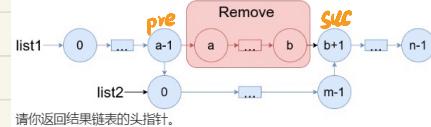
ListNode* mergeTwoLists(ListNode* l1, ListNode* l2) {
    ListNode* tmp = new ListNode();
    ListNode* dummy = tmp;
    while(l1 != NULL && l2 != NULL){
        if(l1->val < l2->val){
            tmp->next = l1;
            l1 = l1->next;
        } else{
            tmp->next = l2;
            l2 = l2->next;
        }
        tmp = tmp->next;
    }
    if(l1 != NULL){
        tmp->next = l1;
    }
    else if(l2 != NULL){
        tmp->next = l2;
    }
    return dummy->next;
}

```

# 2.

给你两个链表 list1 和 list2，它们包含的元素分别为 n 个和 m 个。  
请你将 list1 中下标从 a 到 b 的全部节点删除，并将 list2 接在被删除节点的位置。

下图中蓝色边和节点展示了操作后的结果：



```

ListNode* mergeInBetween(ListNode* list1, int a, int b, ListNode* list2) {
    ListNode* dummy = new ListNode();
    dummy->next = list1;
    ListNode* pre = dummy;
    for(int i = 0; i < a; i++) {pre = pre->next;}
    ListNode* suc = pre->next;
    for(int i = 0; i < b-a+1; i++) {suc = suc->next;}
    pre->next = list2;
    while(list2->next != nullptr) {list2 = list2->next;}
    list2->next = suc;
    return dummy->next;
}

```

# 3.

给你一个二维数组，每个链表都已经按升序排列。  
请你将所有链表合并到一个升序链表中，返回合并后的链表。

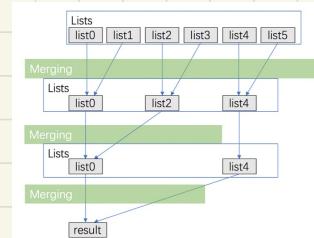
示例 1：

```

输入: lists = [[1,4,5],[1,3,4],[2,6]]
输出: [1,1,2,3,4,4,5,6]
解释: 链表数组如下,
[
  [1->4->5,
  1->3->4,
  2->6
]
将它们合并到一个有序链表中得到。
1->1->2->3->4->4->5->6

```

# 分而治之的思想，类似于归并排序



```

ListNode* mergeKLists(vector<ListNode*>& lists) {
    return merge(lists, 0, (int)lists.size()-1);
}

```

```

ListNode* merge(vector<ListNode*>& lists, int l, int r){
    if(l == r) {return lists[l];}
    if(l > r) {return nullptr;}
    int mid = (l+r)/2;
    return mergeKLists(merge(lists, l, mid), merge(lists, mid+1, r));
}

```

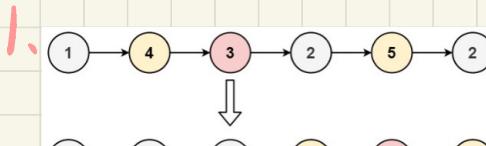
与题目的函数

# 分隔链表

**注明：**这是一个力扣迷惑现象：同名不同题，不同名同题。

1. 86. 分隔链表和面试题 02.04. 分割链表：给定一个链表和一个特定值 x，对链表进行分隔，使得所有小于 x 的节点都在大于或等于 x 的节点之前。

2. 725. 分隔链表：给定一个头结点为 root 的链表，编写一个函数以将链表分隔为 k 个连续的部分。



输入：head = [1,4,3,2,5,2], x = 3  
输出：[1,2,2,4,3,5]

```

ListNode* partition(ListNode* head, int x) {
    ListNode* smalldummy = new ListNode();
    ListNode* small = smalldummy;
    ListNode* largedummy = new ListNode();
    ListNode* large = largedummy;
    while(head != nullptr){
        if(head->val < x){
            small->next = head;
            small = small->next;
        } else{
            large->next = head;
            large = large->next;
        }
        head = head->next;
    }
    large->next = nullptr;
    small->next = largedummy->next;
    return smalldummy->next;
}

```

2.



输入: head = [1,2,3,4,5,6,7,8,9,10], k = 3

输出: [[1,2,3,4],[5,6,7],[8,9,10]]

解释:

输入被分成了几个连续的部分，并且每部分的长度相差不超过 1。

前面部分的长度大于等于后面部分的长度。

```
vector<ListNode*> splitListToParts(ListNode* head, int k) {
    int cnt = 0;
    ListNode* tmp = head;
    while(tmp != nullptr){
        cnt++;
        tmp = tmp->next;
    }
    int num = cnt/k, remain = cnt%k;
    vector<ListNode*> lists(k, nullptr);
    ListNode* cur = head;
    for(int i = 0; i < k; i++){
        if(cur == nullptr) {break;}
        lists[i] = cur;
        int size = num;
        if(i < remain) {size += 1;}
        for(int j = 0; j < size-1; j++) {cur = cur->next;}
        ListNode* suc = cur->next;
        cur->next = nullptr;
        cur = suc;
    }
    return lists;
}
```

# 5. 二叉树

## 创建

```
/*创建二叉树*/
class TreeNode{
public:
    int val;
    TreeNode* left;
    TreeNode* right;
    TreeNode() {this->val = 0; this->left = nullptr; this->right = nullptr;};
    TreeNode(int val) {this->val = val; this->left = nullptr; this->right = nullptr;};
    TreeNode(int val, TreeNode* left, TreeNode* right) {this->val; this->left; this->right;};
};
```

## 遍历

```
/*先序遍历*/
/*递归版*/
vector<int> ans;
void preorderTraversal(TreeNode* cur, vector<int>& ans){
    if(cur == nullptr) {return;};
    ans.emplace_back(cur->val);
    preorderTraversal(cur->left, ans);
    preorderTraversal(cur->right, ans);
}

/*迭代版*/
vector<int> preorderTraversal(TreeNode* root){
    vector<int> ans;
    stack<TreeNode*> st;
    if(root != nullptr) {st.push(root);}
    while(!st.empty()){
        TreeNode* node = st.top();
        if(node != nullptr){
            st.pop();
            if(node->right) {st.push(node->right);}
            if(node->left) {st.push(node->left);}
            st.push(node); st.push(nullptr);
        }
        else{
            st.pop();
            node = st.top();
            st.pop();
            ans.emplace_back(node->val);
        }
    }
    return ans;
}
```

```
/*中序遍历*/
/*递归版*/
vector<int> ans;
void inorderTraversal(TreeNode* cur, vector<int>& ans){
    if(cur == nullptr) {return;};
    preorderTraversal(cur->left, ans);
    ans.emplace_back(cur->val);
    preorderTraversal(cur->right, ans);
}

/*迭代版*/
vector<int> inorderTraversal(TreeNode* root){
    vector<int> ans;
    stack<TreeNode*> st;
    if(root != nullptr) {st.push(root);}
    while(!st.empty()){
        TreeNode* node = st.top();
        if(node != nullptr){
            st.pop();
            if(node->right) {st.push(node->right);}
            st.push(node); st.push(nullptr);
            if(node->left) {st.push(node->left);}
        }
        else{
            st.pop();
            node = st.top();
            st.pop();
            ans.emplace_back(node->val);
        }
    }
    return ans;
}
```



我们以中序遍历为例，无法同时解决访问节点（遍历节点）和处理节点（将元素放进结果集）不一致的情况。

那我们就将访问的节点放入栈中，把要处理的节点也放入栈中但是要做标记。

如何标记呢，就是要处理的节点放入栈之后，紧接着放入一个空指针作为标记。这种方法也可以叫做标记法。

```

/*后序遍历/
/*递归版*/
vector<int> ans;
void postorderTraversal(TreeNode* cur, vector<int>& ans){
    if(cur == nullptr) {return;}
    preorderTraversal(cur->left, ans);
    preorderTraversal(cur->right, ans);
    ans.emplace_back(cur->val);
}
/*迭代版*/
vector<int> postorderTraversal(TreeNode* root){
    vector<int> ans;
    stack<TreeNode*> st;
    if(root != nullptr) {st.push(root);}
    while(!st.empty()){
        TreeNode* node = st.top();
        if(node != nullptr){
            st.pop();
            st.push(node);
            st.push(nullptr);
            if(node->right) {st.push(node->right);}
            if(node->left) {st.push(node->left);}
        }
        else{
            st.pop();
            node = st.top();
            st.pop();
            ans.emplace_back(node->val);
        }
    }
    return ans;
}

```

```

/*层次遍历*/
vector<vector<int>> levelTraversal(TreeNode* root){
    vector<vector<int>> ans;
    queue<TreeNode*> q;
    if(root != nullptr) {q.push(root);}
    while (!q.empty()){
        int size = q.size();
        vector<int> tmp;
        for(int i = 0; i < size; i++){
            TreeNode* node = q.front();
            q.pop();
            tmp.emplace_back(node->val);
            if(node->left) {q.push(node->left);}
            if(node->right) {q.push(node->right);}
        }
        ans.emplace_back(tmp);
    }
    return ans;
}

```

类似BFS

## 二叉树的遍历

**提升：**在本地编译器跑起来，并思考如何将返回值改为void。

1. [144. 二叉树的前序遍历](#)
2. [94. 二叉树的中序遍历](#)
3. [145. 二叉树的后序遍历](#)
4. [剑指 Offer 33. 二叉搜索树的后序遍历序列](#): 输入一个整数数组，判断该数组是不是某二叉搜索树的后序遍历结果。
5. [剑指 Offer 32 - I. 从上到下打印二叉树](#): 自上而下，自左而右，不分层输出
6. [剑指 Offer 32 - II. 从上到下打印二叉树 II](#)和[102. 二叉树的层序遍历](#): 自上而下，自左而右，分层输出
7. [107. 二叉树的层次遍历 II](#): 自下而上，自左而右，分层输出
8. [剑指 Offer 32 - III. 从上到下打印二叉树 III](#)和[103. 二叉树的锯齿形层次遍历](#): 先从左往右，再从右往左进行下一层遍历，以此类推，层与层之间交替进行。

4. **后序遍历定义：**【左子树 | 右子树 | 根节点】，即遍历顺序为“左、右、根”。

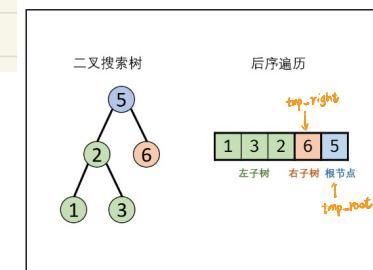
**二叉搜索树定义：**左子树中所有节点的值 < 根节点的值；右子树中所有节点的值 > 根节点的值；其左、右子树也分别为二叉搜索树。

```

bool verifyPostorder(vector<int>& postorder) {
    return judge(postorder, 0, postorder.size()-1);
}

bool judge(vector<int>& postorder, int i, int j){
    if(i >= j) {return true;}
    int tmp = i;
    while(postorder[tmp] < postorder[j]) {tmp++;}
    int tmp_right = tmp;
    while(postorder[tmp] > postorder[j]) {tmp++;}
    int tmp_root = tmp;
    return tmp_root == j && judge(postorder, i, tmp_right-1) && judge(postorder, tmp_right, tmp_root-1);
}

```



## 5.6. 层次遍历

<https://leetcode.cn/problems/binary-tree-level-order-traversal/solution/dai-ma-sui-xiang-lu-er-cha-shu-ceng-xu-b-zhun/>

7. 最后加上 `reverse(ans.begin(), ans.end())` 吧

8. 在 `for` 循环后加上  
`cnt++;`  
`if(cnt%2 == 0) {reverse(tmp.begin(), tmp.end());}`  
`ans.emplace_back(tmp);`

### 构造二叉树

相同点：输入都不是 `TreeNode`，输出都是 `TreeNode`

1. 105. 从前序与中序遍历序列构造二叉树和剑指 Offer 07. 重建二叉树

2. 106. 从中序与后序遍历序列构造二叉树

3. 889. 根据前序和后序遍历构造二叉树

4. 108. 将有序数组转换为二叉搜索树和面试题 04.02. 最小高度树：将一个按照升序排列的有序数组，转换为一棵高度平衡二叉搜索树。

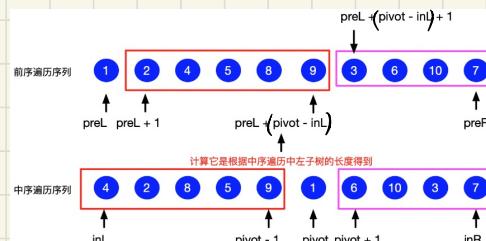
注意此处的最小高度树还有个同名但不相关的题目310. 最小高度树，别搞混了。[\(2\)详见鄙视章节](#)

5. 109. 有序链表转换二叉搜索树：给定一个单链表，其中的元素按升序排序，将其转换为高度平衡的二叉搜索树。

相反题：114. 二叉树展开为链表：给定一个二叉树，原地将它展开为一个单链表。

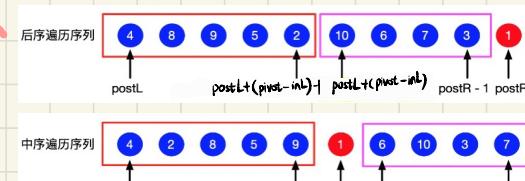
(平衡/搜索)二叉树遍历使用  
递归思想

1. 前序遍历数组的第一个数（索引为 0）的数一定是二叉树的根结点，于是在中序遍历中找这个根结点的索引，然后把“前序遍历数组”和“中序遍历数组”分为两个部分，就分别对应二叉树的左子树和右子树，分别递归完成就可以了。



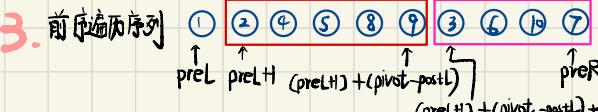
```
TreeNode* buildTree(vector<int>& preorder, vector<int>& inorder) {
    return build(preorder, 0, preorder.size()-1, inorder, 0, inorder.size()-1);
}

TreeNode* build(vector<int>& preorder, int preleft, int preright, vector<int>& inorder, int inleft, int inright) {
    if(preleft > preright || inleft > inright) {return nullptr;}
    int pivot = preorder[preleft];
    TreeNode* root = new TreeNode(pivot);
    int pivotIndex = inleft;
    while(inorder[pivotIndex] != pivot && pivotIndex < inright){
        pivotIndex++;
    }
    root->left = build(preorder, preleft+1, preleft+(pivotIndex-inleft), inorder, inleft, pivotIndex-1);
    root->right = build(preorder, preleft+(pivotIndex-inleft)+1, preright, inorder, pivotIndex+1, inright);
    return root;
}
```



```
TreeNode* buildTree(vector<int>& inorder, vector<int>& postorder) {
    return build(postorder, 0, postorder.size()-1, inorder, 0, inorder.size()-1);
}

TreeNode* build(vector<int>& postorder, int postleft, int postright, vector<int>& inorder, int inleft, int inright) {
    if(postleft > postright || inleft > inright) {return nullptr;}
    int pivot = postorder[postright];
    TreeNode* root = new TreeNode(pivot);
    int pivotIndex = inleft;
    while(inorder[pivotIndex] != pivot && pivotIndex < inright){
        pivotIndex++;
    }
    root->left = build(postorder, postleft, postleft+(pivotIndex-inleft)-1, inorder, inleft, pivotIndex-1);
    root->right = build(postorder, postleft+(pivotIndex-inleft), postright-1, inorder, pivotIndex+1, inright);
    return root;
}
```



```
TreeNode* constructFromPrePost(vector<int>& preorder, vector<int>& postorder) {
    return build(preorder, 0, preorder.size()-1, postorder, 0, postorder.size()-1);
}

TreeNode* build(vector<int>& preorder, int preleft, int preright, vector<int>& postorder, int postleft, int postright) {
    if(preleft > preright || postleft > postright) {return nullptr;}
    int pivot = postorder[postright];
    TreeNode* root = new TreeNode(pivot);
    if(pivot == preleft) {return root;}
    int pivotIndex = postleft;
    int pivotIndex_ = postleft;
    while(postorder[pivotIndex] != pivot &amp; pivotIndex < postright){
        pivotIndex++;
    }
    root->left = build(preorder, preleft+1, (preleft+1)+(pivotIndex-postleft), postorder, postleft, pivotIndex);
    root->right = build(preorder, (preleft+1)+(pivotIndex-postleft)+1, preright, postorder, pivotIndex+1, postright-1);
    return root;
}
```

4.

给你一个整数数组 `nums`，其中元素已经按 **升序** 排列，请你将其转换为一棵 **高度平衡二叉搜索树**。

**高度平衡二叉树**是一棵满足「每个节点的左右两个子树的高度差的绝对值不超过 1」的二叉树。 \*

```
TreeNode* sortedArrayToBST(vector<int>& nums) {
    return build(nums, 0, nums.size()-1);
}

TreeNode* build(vector<int>& nums, int l, int r){
    if(l>r) {return nullptr;}
    int mid = l+(r-l)/2;
    TreeNode* root = new TreeNode(nums[mid]);
    root->left = build(nums, l, mid-1);
    root->right = build(nums, mid+1, r);
    return root;
}
```

## 拓展: 题654最大二叉树

给定一个不重复的整数数组 `nums`。**最大二叉树**可以用下面的算法从 `nums` 递归地构建：

1. 创建一个根节点，其值为 `nums` 中的最大值。
2. 递归地在最大值 **左边** 的 **子数组前缀上** 构建左子树。
3. 递归地在最大值 **右边** 的 **子数组后缀上** 构建右子树。

返回 `nums` 构建的 **最大二叉树**。

```
TreeNode* constructMaximumBinaryTree(vector<int>& nums) {
    return build(nums, 0, nums.size()-1);
}

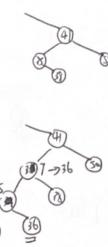
TreeNode* build(vector<int>& nums, int l, int r){
    if(l>r) {return nullptr;}
    int maxval = INT_MIN, maxindex = 0;
    for(int i = l; i<=r; i++){
        if(nums[i] > maxval) {
            maxval = nums[i];
            maxindex = i;
        }
    }
    TreeNode* root = new TreeNode(maxval);
    root->left = build(nums, l, maxindex-1);
    root->right = build(nums, maxindex+1, r);
    return root;
}
```

## 拓展: 二叉搜索树相关操作

```
/*二叉搜索树查找*/
TreeNode* find(TreeNode* root, int val){
    if(root == nullptr) {return nullptr;}
    if(val < root->val) {return find(root->left, val);}
    else if(val > root->val) {return find(root->right, val);}
    else {return root;}
}
```

```
/*二叉搜索树插入*/
TreeNode* insert(TreeNode* root, int val){
    if(root == nullptr) {
        TreeNode* node = new TreeNode(val);
        return node;
    }
    if(val < root->val) {root->left = insert(root->left, val);}
    else if(val > root->val) {root->right = insert(root->right, val);}
    return root;
}
```

- ① 要删除结点为叶结点：直接删除。  
 ② 要删除结点有一个儿子：删除 35 后，直接把 38 搬到 41 的左边（升左子树的值都小于 41）  
 41 的树值都大于 41。  
 ③ 要删除结点有左右子树：寻找 31 的右子树，  
 最值或右子树最小值，将其代替 31，  
 而 36/38 只能有一个没有儿子，则又回到  
 ①②的情况再来删除 34/38。



```
/*二叉搜索树寻找最小值*/
TreeNode* findmin(TreeNode* root){
    if(root == nullptr) {return nullptr;}
    else if(root->left == nullptr) {return root;}
    else {return findmin(root->left);}
}

/*二叉搜索树寻找最大值*/
TreeNode* findmax(TreeNode* root){
    if(root == nullptr) {return nullptr;}
    else if(root->right == nullptr) {return root;}
    else {return findmax(root->right);}
}
```

```
/*二叉搜索树删除结点*/
TreeNode* deleteNode(TreeNode* root, int key) {
    if(root == nullptr) {return nullptr;} //未找到此结点
    else if(key < root->val){//递归到左子树
        root->left = deleteNode(root->left, key);
        return root;
    }
    else if(key > root->val){//递归到右子树
        root->right = deleteNode(root->right, key);
        return root;
    }
    else{//找到此结点
        if(root->left && root->right){//被删除结点有左右结点，对应情况3
            TreeNode *successor = root->right;
            while(successor->left) //找到右子树最小结点(也可以找左子树最大结点)
                successor = successor->left;
            root->right = deleteNode(root->right, successor->val);
            successor->right = root->right;
            successor->left = root->left;//此结点移动到被删除的结点的位置
            return successor;
        }
        //被删除结点有一个或没有子结点，对应情况2或1
        else if(root->right == nullptr) {return root->left;}
        else if(root->left == nullptr) {return root->right;}
        else {return nullptr;}
    }
}
```

# 5. 题4是数组 $\Rightarrow$ 二叉树

## 题5是链表 $\Rightarrow$ 二叉树

```

TreeNode* sortedListToBST(ListNode* head) {
    return build(head, nullptr);
}

TreeNode* build(ListNode* l, ListNode* r){
    if(l == r) {return nullptr;}
    ListNode* mid = findmid(l, r);
    TreeNode* root = new TreeNode(mid->val);
    root->left = build(l, mid);
    root->right = build(mid->next, r);
    return root;
}

快慢指针找中点的变形
ListNode* findmid(ListNode* l, ListNode* r){
    ListNode *slow = l, *fast = l;
    while(fast != r && fast->next != r){
        slow = slow->next;
        fast = fast->next->next;
    }
    return slow;
}

```

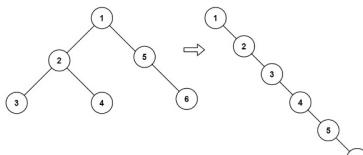
(其实也可以遍历链表 - 次转为数组，  
此时又回到题4的解法)

# 5的相反题。

给你二叉树的根节点  $root$ ，请你将它展开为一个单链表：

- 展开后的单链表应该同样使用  $TreeNode$ ，其中  $right$  子指针指向链表中下一个结点，而左子指针始终为  $null$ 。
- 展开后的单链表应该与二叉树先序遍历顺序相同。

示例1：



展开为单链表的做法是，维护上一个访问的节点  $prev$ ，每次访问一个节点时，令当前访问的节点为  $curr$ ，将  $prev$  的左子节点设为  $null$  以及将  $prev$  的右子节点设为  $curr$ ，然后将  $curr$  赋值给  $prev$ ，进入下一个节点的访问，直到遍历结束。需要注意的是，初始时  $prev$  为  $null$ ，只有在  $prev$  不为  $null$  时才能对  $prev$  的左子节点进行更新。

### 迭代版先序遍历的变形

#### 变形

(其实还有另一种方法  
是先序遍历并  
用一个vector记录  
所有TreeNode，  
再依次接到root上)

```

void flatten(TreeNode* root) {
    stack<TreeNode*> st;
    if(root != nullptr) {st.push(root);}
    TreeNode* pre = nullptr;
    while(!st.empty()){
        TreeNode* node = st.top();
        if(node != nullptr){
            st.pop();
            if(node->right) {st.push(node->right);}
            if(node->left) {st.push(node->left);}
            st.push(node); st.push(nullptr);
        } else{
            st.pop();
            node = st.top();
            st.pop();
            if(pre != nullptr){
                pre->left = nullptr;
                pre->right = node;
            }
            pre = node; // 更新pre
        }
    }
}

```

} 将 pre 与 node 相连

## 输出二叉树

### 1. 655. 输出二叉树：按一定规则逐层输出

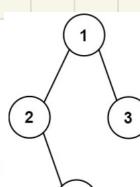
### 2. 剑指 Offer 37. 序列化二叉树和 297. 二叉树的序列化与反序列化：String 和 TreeNode 的相互转换



给你一棵二叉树的根节点  $root$ ，请你构造一个下标从 0 开始、大小为  $m \times n$  的字符串矩阵  $res$ ，用以表示树的格式化布局。构造此格式化布局矩阵需要遵循以下规则：

- 树的高度为  $height$ ，矩阵的行数  $m$  应该等于  $height + 1$ 。
- 矩阵的列数  $n$  应该等于  $2^{height+1} - 1$ 。
- 根节点需要放置在顶行的正中间，对应位置为  $res[0][n-1/2]$ 。
- 对于放置在矩阵中的每个节点，设对应位置为  $res[r][c]$ ，将其左子节点放置在  $res[r+1][c-2^{height-r-1}]$ ，右子节点放置在  $res[r+1][c+2^{height-r-1}]$ 。
- 继续这一过程，直到树中的所有节点都妥善放置。
- 任意空单元格都应该包含空字符串 ""。

返回构造得到的矩阵  $res$ 。



输出：root = [1,2,3,null,4]  
输出：  
[[ "", "", "", "1", "", "", "" ],  
 [ "", "2", "", "", "3", "", "" ],  
 [ "", "", "4", "", "", "", "" ]]

```

vector<vector<string>> printTree(TreeNode* root) {
    int m = getheight(root); // 行
    int n = pow(2, m)-1; // 列
    chess(m, vector<string>(n, ""));
    printchess(root, 0, n-1, 0, chess);
    return chess;
}

int getheight(TreeNode* root){
    if(root == nullptr) {return 0;}
    return max(getheight(root->left), getheight(root->right))+1;
}

void printchess(TreeNode* root, int l, int r, int row, vector<vector<string>>& chess){
    if(root == nullptr) {return;}
    int mid = (l+r)/2;
    chess[row][mid] = to_string(root->val);
    printchess(root->left, l, mid-1, row+1, chess);
    printchess(root->right, mid+1, r, row+1, chess);
    return;
}

```

} 二叉树的高度获取 (DFS的思想)

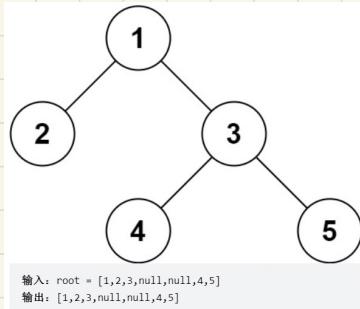
} 二分+分治的思想

2.

请实现两个函数，分别用来序列化和反序列化二叉树。

你需要设计一个算法来实现二叉树的序列化与反序列化。这里不限定你的序列 / 反序列化算法执行逻辑，你只需要保证一个二叉树可以被序列化为一个字符串并且将这个字符串反序列化为原始的树结构。

**提示：**输入输出格式与 LeetCode 目前使用的方式一致，详情请参阅 LeetCode 序列化二叉树的格式。你并非必须采取这种方式，你也可以采用其他的方法解决这个问题。



```

// Encodes a tree to a single string.
string serialize(TreeNode* root) {
    string ans = "";
    if(root == NULL) {ans = "["; return ans;}
    ans += "[";
    queue<TreeNode*> q;
    q.push(root);
    while(!q.empty()){
        TreeNode* node = q.front();
        q.pop();
        if(node == NULL) {ans += "null,";}
        else{
            ans += to_string(node->val); ans += ",";
            q.push(node->left);
            q.push(node->right);
        }
    }
    ans.pop_back(); ans += "]";
    return ans;
}
  
```

```

// Decodes your encoded data to tree.
TreeNode* deserialize(string data) {
    if(data == "[") {return NULL;}
    vector<string> tree_data;
    string s = "";
    for(auto& c:data){
        if(c == '[') {continue;}
        else if(c == ']') {tree_data.emplace_back(s); s = "";}
        else if(c == ',') {tree_data.emplace_back(s); s = "";}
        else {s += c;}
    }
    queue<TreeNode*> q;
    TreeNode* root = new TreeNode(stoi(tree_data[0]));
    q.push(root);
    int cnt = 1;
    while(!q.empty()){
        TreeNode* node = q.front();
        q.pop();
        if(tree_data[cnt] != "null"){
            node->left = new TreeNode(stoi(tree_data[cnt]));
            q.push(node->left);
        }
        cnt++;
        if(tree_data[cnt] != "null"){
            node->right = new TreeNode(stoi(tree_data[cnt]));
            q.push(node->right);
        }
        cnt++;
    }
    return root;
}
  
```

## 二叉树的验证

注明：有多个类似题的是重点题目，其他的量力而行。

100. 相同的树：检验是否相同。
- 剑指 Offer 28. 对称的二叉树和101. 对称二叉树：判断一棵二叉树是不是对称的
- 剑指 Offer 55 - II. 平衡二叉树、110. 平衡二叉树和面试题 04.04. 检查平衡性：判断该树是不是平衡二叉树。如果某二叉树中任意节点的左右子树的深度相差不超过1，那么它就是一棵平衡二叉树
98. 验证二叉搜索树和面试题 04.05. 合法二叉搜索树：判断其是否是一个有效的二叉搜索树
1361. 验证二叉树：只有所有节点能够形成且只形成一颗有效的二叉树时，返回true；否则返回false。
958. 二叉树的完全性检验：确定它是否是一个完全二叉树。
993. 二叉树的堂兄弟节点：判断x和y是否为堂兄弟
- 剑指 Offer 26. 树的子结构：判断B是不是A的子结构
572. 另一个树的子树和面试题 04.10. 检查子树：判断T2是否为T1的子树

1.

给你两棵二叉树的根节点 p 和 q，编写一个函数来检验这两棵树是否相同。

如果两个树在结构上相同，并且节点具有相同的值，则认为它们是相同的。

递归

```
bool judge(TreeNode* p, TreeNode* q){  
    if(p == nullptr && q == nullptr) {return true;}  
    if(p == nullptr || q == nullptr) {return false;}  
    if(p->val == q->val) {return judge(p->left, q->left) && judge(p->right, q->right);}  
    else {return false;}  
}
```

2.

请实现一个函数，用来判断一棵二叉树是不是对称的。如果一棵二叉树和它的镜像一样，那么它是对称的。

例如，二叉树 [1,2,2,3,4,4,3] 是对称的。

```
1  
/\ \/  
2 2  
/\ / \/  
3 4 4 3
```

```
bool isSymmetric(TreeNode* root) {  
    if(root == NULL) {return true;}  
    return judge(root->left, root->right);  
}  
  
bool judge(TreeNode* p, TreeNode* q){  
    if(p == NULL && q == NULL) {return true;}  
    if(p == NULL || q == NULL) {return false;}  
    if(p->val == q->val) {return judge(p->left, q->right) && judge(p->right, q->left);}  
    else {return false;}  
}
```

3. ①

```
bool isBalanced(TreeNode* root) {  
    if(root == NULL) {return true;}  
    int h1 = getheight(root->left);  
    int h2 = getheight(root->right);  
    if((h1-h2)>1 || (h1-h2)<-1) {return false;}  
    return isBalanced(root->left) && isBalanced(root->right);  
}  
  
int getheight(TreeNode* root){  
    if(root == NULL) {return 0;}  
    return max(getheight(root->left), getheight(root->right))+1;  
}
```

迭代

```
bool judge(TreeNode* u, TreeNode* v){  
    queue <TreeNode*> q;  
    q.push(u); q.push(v);  
    while (!q.empty()) {  
        u = q.front(); q.pop();  
        v = q.front(); q.pop();  
        if (!u && !v) continue;  
        if ((!u || !v) || (u->val != v->val)) return false;  
  
        q.push(u->left);  
        q.push(v->left);  
  
        q.push(u->right);  
        q.push(v->right);  
    }  
    return true;  
}
```

②提前阻断

• 问题描述：  
1. 当执行 isBalanced() 时，右子树的遍历深度 < 1，说明以结点 root 为根的左子树的最大深度 root->right->max(left, right)+1；  
2. 当执行 root 左/右子树的遍历深度 > 2，说明 -1，代表 **该子树不平衡树**。  
• 请回答问题：  
1. 当结点叶子时，返回深度 0；  
2. 当结点 (右) 子树遍历 (left=-1) 时，代表右子树的 (右) 子树 不是平衡树，直接返回 -1；

```
bool isBalanced(TreeNode* root) {  
    int f = judge(root);  
    if(f == -1) {return false;}  
    else {return true;}  
}  
  
int judge(TreeNode* root){  
    if(root == nullptr) {return 0;}  
    int l = judge(root->left);  
    if(l == -1) {return -1;}  
    int r = judge(root->right);  
    if(r == -1) {return -1;}  
    if((l-r)>1 || (l-r)<-1) {return -1;}  
    else {return max(l, r)+1;}  
}
```

## 4. ① 填空

```

bool isValidBST(TreeNode* root) {
    return judge(root, LONG_MIN, LONG_MAX);
}

bool judge(TreeNode* root, long min, long max){
    if(root == nullptr) {return true;}
    if(root->val <= min || root->val >= max) {return false;}
    return judge(root->left, min, root->val) && judge(root->right, root->val, max);
}

```

※ 只要二叉树中序遍历后是按升序排列的，  
那么二叉树就是二叉搜索树

## ② 中序遍历变形

```

bool isValidBST(TreeNode* root) {
    if(root == nullptr) {return true;}
    stack<TreeNode*> st;
    long pre = LONG_MIN;
    st.push(root);
    while(!st.empty()){

        TreeNode* node = st.top();
        if(node != nullptr){
            st.pop();
            if(node->right) {st.push(node->right);}
            st.push(node);
            st.push(nullptr);
            if(node->left) {st.push(node->left);}
        }
        else{
            st.pop();
            node = st.top();
            st.pop();
            if(node->val > pre) {pre = node->val;}
            else {return false;}
        }
    }
    return true;
}

```

## 5.

二叉树上有  $n$  个节点，按从 0 到  $n - 1$  编号，其中节点  $i$  的两个子节点分别是  $\text{leftChild}[i]$  和  $\text{rightChild}[i]$ 。

只有所有节点能够形成且只形成一颗有效的二叉树时，返回 `true`；否则返回 `false`。

如果节点  $i$  没有左子节点，那么  $\text{leftChild}[i]$  就等于  $-1$ 。右子节点也符合该规则。

## ① 并查集

(1) 初始化  $n$  个连通集，让每个结点的祖宗结点是它本身  
(2) 设要加的边是  $a \rightarrow b$ ，我们判断下。如果  $a$  和  $b$  的祖宗相等，说明你加上这条边会形成环，返回 `false`，如果不相等，继续往下判断  
(3) 因为这里是单向边，如果像示例 2 的结点 3 一样有两个结点指向它，也是不成立的。所以我们在 (2) 的基础上还有判断箭头指向的结点  $b$  原来是否有其他结点指向它，如果有则返回 `false`，否则将  $b$  加到  $a$  所在的连通集中，即  $\text{p}[pb] = pa$   
(4) 最后判断连通块的个数是否为 1 即可

```

bool validateBinaryTreeNodes(int n, vector<int>& leftChild, vector<int>& rightChild) {
    UnionFind* uf = new UnionFind(n);
    for(int i = 0; i < n; i++){
        int l = leftChild[i];
        if(l != -1){
            int pi = find_parents(uf, i), pl = find_parents(uf, l);
            if(pi == pl) {return false;} // 如果父母相等，说明加上这条边会形成环
            else{
                if(pi == 1) {union_nodes(uf, i, l);}
                else{if(pi == pl) {return false;} // 箭头指向的结点已经有别的结点指向了
                }
            }
        }
        int r = rightChild[i];
        if(r != -1){
            int pi = find_parents(uf, i), pr = find_parents(uf, r);
            if(pi == pr) {return false;} // 如果父母相等，说明加上这条边会形成环
            else{
                if(pr == r) {union_nodes(uf, i, r);}
                else {return false;} // 箭头指向的结点已经有别的结点指向了
            }
        }
    }
    int cnt = 0;
    for(int i = 0; i < n; i++){
        if(uf->parents[i] == i) {cnt++;}
    }
    return cnt == 1;
}

```

## 7.

在二叉树中，根节点位于深度 0 处，每个深度为  $k$  的节点的子节点位于深度  $k+1$  处。

如果二叉树的两个节点深度相同，但父节点不同，则它们是一对堂兄弟节点。

我们给出了具有唯一值的二叉树的根节点 `root`，以及树中两个不同节点的值 `x` 和 `y`。

只有与值 `x` 和 `y` 对应的节点是堂兄弟节点时，才返回 `true`。否则，返回 `false`。

## ③ BFS

```

bool validateBinaryTreeNodes(int n, vector<int>& leftChild, vector<int>& rightChild) {
    vector<int> indeg(n, 0); // 统计入度
    for(int i = 0; i < n; i++){
        if(leftChild[i] != -1) {indeg[leftChild[i]]++;}
        if(rightChild[i] != -1) {indeg[rightChild[i]]++;}
    }
    int root = -1;
    for(int i = 0; i < n; i++){
        if(indeg[i] == 0) {root = i; break;} // 找到一个入度为 0 的根节点
    }
    if(root == -1) {return false;}
    int cnt = 0;
    queue<int> q;
    vector<int> visit(n, 0);
    q.push(root);
    visit[root]++;
    while(!q.empty()){
        int node = q.front(); q.pop();
        cnt++;
        if(leftChild[node] != -1){
            visit[leftChild[node]]++;
            if(visit[leftChild[node]] > 1) {return false;}
            else {q.push(leftChild[node]);}
        }
        if(rightChild[node] != -1){
            visit[rightChild[node]]++;
            if(visit[rightChild[node]] > 1) {return false;}
            else {q.push(rightChild[node]);}
        }
    }
    return cnt == n;
}

```

## ④ 子结构

输入两个二叉树 A 和 B，判断 B 是不是 A 的子结构。(约定空树不算是一个树的子结构)

B 是 A 的子结构，即 A 中有出现和 B 相同的结构和节点值。

例如：

给定的树 A：

3

/ \

4 5

/ \

1 2

给定的树 B：

4

/

1

返回 `true`，因为 B 与 A 的一个子树拥有相同的结构和节点值。

## 8.

```

bool isCousins(TreeNode* root, int x, int y) {
    queue<TreeNode*> q;
    if(root != nullptr) {q.push(root);}
    else{return false;}
    int cnt = -1;
    vector<TreeNode*> p(2, nullptr);
    while(!q.empty()){
        int size = q.size();
        for(int i = 0; i < size; i++){
            TreeNode* node = q.front(); q.pop();
            if(node->left){
                if(node->left->val == x || node->left->val == y){
                    cnt++; p[cnt] = node;/cnt加一, 记录其父节点
                }
                else {q.push(node->left);}
            }
            if(node->right){
                if(node->right->val == x || node->right->val == y){
                    cnt++; p[cnt] = node;/cnt加一, 记录其父节点
                }
                else {q.push(node->right);}
            }
        }
        if(cnt == 1 && p[0] != p[1]) {return true;}//如果找到x和y, 且其父结点不同
    }
    cnt = -1;/下一次循环便开始遍历新的一层, cnt还原
}
return false;
}

```

## 9、子树

给定两个二叉树 `root` 和 `subRoot`。检验 `root` 中是否包含和 `subRoot` 具有相同结构和节点值的子树。如果存在，返回 `true`；否则，返回 `false`。

二叉树 `tree` 的一棵子树包括 `tree` 的某个节点和这个节点的所有后代节点。`tree` 也可以看做它自身的一棵子树。

## 方法：题8的前序遍历+题10的函数

```

bool isSubStructure(TreeNode* A, TreeNode* B) {
    if(B == NULL) {return false;}
    stack<TreeNode*> st;
    if(A != NULL) {st.push(A);}
    while(!st.empty()){
        TreeNode* node = st.top();
        if(node != NULL){
            st.pop();
            if(node->right) {st.push(node->right);}
            if(node->left) {st.push(node->left);}
            st.push(node); st.push(NULL);
        }
        else{
            st.pop();
            node = st.top();
            st.pop();
            if(node->val == B->val){
                if(judge(node, B)) {return true;}
            }
        }
    }
    return false;
}

```

⇒题8的变形  
`bool judge(TreeNode* node, TreeNode* B){`  
`if(B == NULL) {return true;}`  
`if(node == NULL) {return false;}`  
`if(node->val == B->val) {return judge(node->left, B->left) && judge(node->right, B->right);}`  
`else{return false;}}`

## 6、完全二叉树

在一个 `完全二叉树` 中，除了最后一个关卡外，所有关卡都是完全被填满的，并且最后一个关卡中的所有节点都是尽可能靠左的。它可以包含 1 到  $2^h$  节点之间的最后一级 `h`。

```

bool isCompleteTree(TreeNode* root) {
    if(root == nullptr) {return false;}
    queue<TreeNode*> q;
    q.push(root);
    bool flag = false; //true时其左右子结点不全
    while(!q.empty()){
        TreeNode* node = q.front();
        q.pop();
        TreeNode* l = node->left, *r = node->right;
        if((flag && (l != nullptr || r != nullptr)) || (l == nullptr && r != nullptr))
            return false;
        }
        if(l != nullptr) {q.push(l);}
        if(r != nullptr) {q.push(r);}
        if(l == nullptr || r == nullptr) {flag = true;}
    }
    return true;
}

```

## 二叉树的翻转

1. 剑指 Offer 27. 二叉树的镜像和226. 翻转二叉树：左右对称翻转
2. 156.上下翻转二叉树：上下翻转

## I. 递归版

```

TreeNode* mirrorTree(TreeNode* root) {
    build(root);
    return root;
}

void build(TreeNode* root) {
    if(root == NULL) {return;}
    {
        TreeNode* tmp = root->left;
        root->left = root->right;
        root->right = tmp;
        build(root->left);
        build(root->right);
    }
}

```

## 迭代版

```

TreeNode* mirrorTree(TreeNode* root) {
    if(root == NULL) {return root;}
    queue<TreeNode*> q;
    q.push(root);
    while(!q.empty()){
        TreeNode* node = q.front();
        q.pop();
        {
            TreeNode* tmp = node->left;
            node->left = node->right;
            node->right = tmp;
            if(node->left) {q.push(node->left);}
            if(node->right) {q.push(node->right);}
        }
    }
    return root;
}

```

## 2.

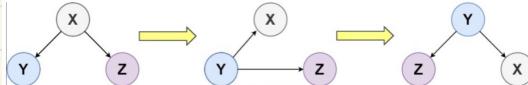
给你一个二叉树的根节点 root，请你将此二叉树上下翻转，并返回新的根节点。

你可以按下面的步骤翻转一棵二叉树：

原来的左子节点变成新的根节点

原来的根节点变成新的右子节点

原来的右子节点变成新的左子节点



树中的每个右节点都有一个同级节点（即共享同一父节点的左节点）

树中的每个右节点都没有子节点

TreeNode\* upsideDownBinaryTree(TreeNode\* root) {

/\*  
因为题目保证了每个有节点都有一个共享父节点的左节点，且没有子节点  
因此，实际上翻转二叉树的方法就是：

1. 对于每一个右节点，将它的兄弟节点的左指针指向它

2. 对于每一个左节点，它的右指针指向自己的父节点

3. 旋转（当然编程时不用体现）

注意，这里只要遍历左子树就行了，因为是二叉树且每一个节点的右孩子都是叶子节点。

\*/

TreeNode\* right = nullptr;

TreeNode\* father = nullptr;

while(root) {

    TreeNode\* left = root->left; // 记录当前root的左孩子

    root->left = right; // 当前节点的左孩子即为父节点的右孩子（如果root节点是第一个

**剪枝** right = root->right; // 记录当前root的右孩子

    root->right = father; // 当前root的右孩子变成原来的父节点

**更新** father = root; // 记录root为father用于后续遍历

    root = left; // root改为刚才保存的root的左孩子（即向下遍历）

}

return father;

## 二叉树的深度和直径

1. 剑指 Offer 55 - I. 二叉树的深度和104. 二叉树的最大深度：找出其最大深度。二叉树的深度为根节点到最近叶子节点的最长路径上的节点数。

→上文已写出

2. 111. 二叉树的最小深度：找出其最小深度。最小深度是从根节点到最近叶子节点的最短路径上的节点数量。

3. 543. 二叉树的直径：计算直径长度。一棵二叉树的直径长度是任意两个结点路径长度中的最大值。这条路径可能穿过也可能不穿过根结点。

## 2. ① DFS

```
int minDepth(TreeNode* root) {
    int mind = INT_MAX, d = 0;
    if(root == nullptr) {return d;}
    else {d++;}
    dfs(root, d, mind);
    return mind;
}

void dfs(TreeNode* root, int d, int& mind){
    if(root == nullptr) {return;}
    if(root->left == nullptr && root->right == nullptr){
        mind = min(d, mind);
    }
    if(root->left) {dfs(root->left, d+1, mind);}
    if(root->right) {dfs(root->right, d+1, mind);}
}
```

## ② BFS(层次遍历变形)

```
int minDepth(TreeNode* root) {
    int d = 0;
    if(root == nullptr) {return d;}
    queue<TreeNode*> q;
    q.push(root);
    while(!q.empty()){
        d++;
        int size = q.size();
        for(int i = 0; i<size; i++){
            TreeNode* node = q.front();
            q.pop();
            if(node->left == nullptr && node->right == nullptr){
                return d;
            }
            if(node->left) {q.push(node->left);}
            if(node->right) {q.push(node->right);}
        }
    }
    return d;
}
```

## 3. 求二叉树的高度的变形

```
int diameterOfBinaryTree(TreeNode* root) {
    int maxd = 0;
    dfs(root, maxd);
    return maxd-1; // maxd实际算的是结点个数，则距离要减去1
}

int dfs(TreeNode* root, int& maxd){
    if(root == nullptr) {return 0;}
    int left = dfs(root->left, maxd);
    int right = dfs(root->right, maxd);
    maxd = max(maxd, left+right+1); // 后者是此时结点的左右子树的高之和，再加上它本身
    return max(left,right)+1;
}
```

## 二叉树的路径与路径和

1. 257. 二叉树的所有路径：给定一个二叉树，返回所有从根节点到叶子节点的路径。
2. 112. 路径总和：给定一个二叉树和一个目标和，判断该树中是否存在根节点到叶子节点的路径，这条路径上所有节点值相加等于目标和。
3. 113. 路径总和 II 和剑指 Offer 34. 二叉树中和为某一值的路径：给定一个二叉树和一个目标和，找到所有从根节点到叶子节点路径总和等于给定目标和的路径。
4. 437. 路径总和 III 和面试题 04.12. 求和路径：打印节点数值总和等于某个给定值的所有路径的数量。注意，路径不一定非得从二叉树的根节点或叶节点开始或结束，但是其方向必须向下(只能从父节点指向子节点方向)。
5. 687. 最长同值路径：给定一个二叉树，找到最长的路径，这个路径中的每个节点具有相同值。这条路径可以经过也可以不经过根节点。

## 1. ①DFS

```
vector<string> binaryTreePaths(TreeNode* root) {  
    vector<string> ans;  
    vector<int> path;  
    if(root == nullptr) {return ans;}  
    dfs(root, path, ans);  
    return ans;  
}  
  
void dfs(TreeNode* root, vector<int>& path, vector<string>& ans){  
    path.emplace_back(root->val);  
    if(root->left == nullptr && root->right == nullptr){  
        string s = "";  
        for(int i = 0; i < path.size(); i++){  
            s += to_string(path[i]);  
            s += "->";  
        }  
        s.pop_back(); s.pop_back();  
        ans.emplace_back(s);  
        return;  
    }  
    if(root->left) {dfs(root->left, path, ans); path.pop_back();}  
    if(root->right) {dfs(root->right, path, ans); path.pop_back();}  
    return;  
}
```

## ②BFS 一般BFS解题步骤得开个queue用广深队列

```
vector<string> binaryTreePaths(TreeNode* root) {  
    vector<string> ans;  
    if(root == nullptr) {return ans;}  
    queue<TreeNode*> q;  
    queue<string> qs;  
    q.push(root);  
    qs.push(to_string(root->val));  
    while(!q.empty()){  
        TreeNode* node = q.front();  
        q.pop();  
        string s = qs.front();  
        qs.pop();  
        if(node->left == nullptr && node->right == nullptr){  
            ans.emplace_back(s);  
        }  
        if(node->left){  
            q.push(node->left);  
            qs.push(s + "->" + to_string(node->left->val));  
        }  
        if(node->right){  
            q.push(node->right);  
            qs.push(s + "->" + to_string(node->right->val));  
        }  
    }  
    return ans;  
}
```

## 2. ①DFS

```
bool hasPathSum(TreeNode* root, int targetSum) {  
    bool ans = false;  
    if(root == nullptr) {return ans;}  
    dfs(root, ans, root->val, targetSum);  
    return ans;  
}  
  
void dfs(TreeNode* root, bool& ans, int sum, int targetSum){  
    if(root->left == nullptr && root->right == nullptr && sum == targetSum) {ans = true;}  
    if(root->left) {dfs(root->left, ans, sum + root->left->val, targetSum);}  
    if(root->right) {dfs(root->right, ans, sum + root->right->val, targetSum);}  
}
```

## ②BFS

```
bool hasPathsum(TreeNode* root, int targetSum) {  
    bool ans = false;  
    if(root == nullptr) {return ans;}  
    queue<TreeNode*> q;  
    queue<int> qsum;  
    q.push(root);  
    qsum.push(root->val);  
    while(!q.empty()){  
        TreeNode* node = q.front();  
        q.pop();  
        int sum = qsum.front();  
        qsum.pop();  
        if(sum == targetSum && node->left == nullptr && node->right == nullptr){  
            return true;  
        }  
        if(node->left){  
            q.push(node->left);  
            qsum.push(sum + node->left->val);  
        }  
        if(node->right){  
            q.push(node->right);  
            qsum.push(sum + node->right->val);  
        }  
    }  
    return false;  
}
```

### 3. ① DFS

```

vector<vector<int>> pathSum(TreeNode* root, int targetSum) {
    vector<vector<int>> ans;
    vector<vector<int>> path;
    if(root == nullptr) {return ans;}
    dfs(root, path, ans, root->val, targetSum);
    return ans;
}

void dfs(TreeNode* root, vector<int>& path, vector<vector<int>>& ans, int sum, int targetSum){
    path.emplace_back(root->val);
    if(root->left == nullptr && root->right == nullptr && sum == targetSum){
        ans.emplace_back(path);
        return;
    }
    if(root->left) {
        dfs(root->left, path, ans, sum + root->left->val, targetSum);
        path.pop_back();
    }
    if(root->right) {
        dfs(root->right, path, ans, sum + root->right->val, targetSum);
        path.pop_back();
    }
}

```

### ② BFS

```

vector<vector<int>> pathSum(TreeNode* root, int targetSum) {
    vector<vector<int>> ans;
    if(root == nullptr) {return ans;}
    queue<TreeNode*> q;
    queue<int> qsum;
    queue<vector<int>> qpath;
    q.push(root);
    qsum.push(root->val);
    qpath.push({root->val});
    while(!q.empty()){
        TreeNode* node = q.front();
        q.pop();
        int sum = qsum.front();
        qsum.pop();
        vector<int> path = qpath.front();
        qpath.pop();
        if(sum == targetSum && node->left == nullptr && node->right == nullptr){
            ans.emplace_back(path);
        }
        if(node->left){
            q.push(node->left);
            qsum.push(sum + node->left->val);
            path.emplace_back(node->left->val);
            qpath.push(path);
            path.pop_back();
        }
        if(node->right){
            q.push(node->right);
            qsum.push(sum + node->right->val);
            path.emplace_back(node->right->val);
            qpath.push(path);
            path.pop_back();
        }
    }
    return ans;
}

```

### 4. ① 前序+DFS

```

int pathSum(TreeNode* root, int targetSum) {
    int ans = 0;
    preordertraversal(root, ans, targetSum);
    return ans;
}

void preordertraversal(TreeNode* root, int& ans, int targetSum){
    if(root == nullptr) {return;}
    dfs(root, root->val, ans, targetSum);
    preordertraversal(root->left, ans, targetSum);
    preordertraversal(root->right, ans, targetSum);
}

void dfs(TreeNode* root, long sum, int& ans, int targetSum){
    if(sum == targetSum) {ans++;}
    if(root->left) {dfs(root->left, sum + root->left->val, ans, targetSum);}
    if(root->right) {dfs(root->right, sum + root->right->val, ans, targetSum);}
}

```

### ② 前序 + BFS

```

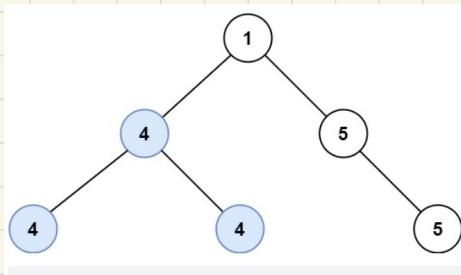
int pathSum(TreeNode* root, int targetSum) {
    int ans = 0;
    preorderdtraversal(root, ans, targetSum);
    return ans;
}

void preorderdtraversal(TreeNode* root, int& ans, int targetSum){
    if(root == nullptr) {return;}
    ans += bfs(root, targetSum);
    preorderdtraversal(root->left, ans, targetSum);
    preorderdtraversal(root->right, ans, targetSum);
}

int bfs(TreeNode* root, int targetSum){
    int cnt = 0;
    if(root == nullptr) {return cnt;}
    queue<TreeNode*> q;
    queue<long> qsum;
    q.push(root);
    qsum.push(root->val);
    while(!q.empty()){
        TreeNode* node = q.front();
        q.pop();
        long sum = qsum.front();
        qsum.pop();
        if(sum == targetSum){
            cnt++;
        }
        if(node->left){
            q.push(node->left);
            qsum.push(sum + node->left->val);
        }
        if(node->right){
            q.push(node->right);
            qsum.push(sum + node->right->val);
        }
    }
    return cnt;
}

```

5.



输入: root = [1,4,5,4,4,5]  
输出: 2

```

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

int dfs(TreeNode* root, int& ans){
    if(root == nullptr) {return 0;}
    int Lcnt = 0, Rcnt = 0, maxcnt = 0;
    int left = dfs(root->left, ans); //root左子树的最长同值路径
    int right = dfs(root->right, ans); //root右子树的最长同值路径
    if(root->left != nullptr && root->right != nullptr && root->left->val == root->val && root->right->val == root->val){//考虑左结点到root到右结点会不会存在最长同值路径
        ans = max(ans, left + right + 2);
    }
    //从左右子树中选择最长的同值路径
    if(root->left != nullptr && root->left->val == root->val) {Lcnt = left + 1;}
    if(root->right != nullptr && root->right->val == root->val) {Rcnt = right + 1;}
    maxcnt = max(Lcnt, Rcnt);
    ans = max(ans, maxcnt);
    return maxcnt;
}

```



## 二叉树的祖先

- 剑指 Offer 68 - I. 二叉搜索树的最近公共祖先和235. 二叉搜索树的最近公共祖先: 给定一个二叉搜索树, 找到该树中两个指定节点的最近公共祖先。
- 剑指 Offer 68 - II. 二叉树的最近公共祖先和236. 二叉树的最近公共祖先: 给定一个二叉树, 找到该树中两个指定节点的最近公共祖先。

6

1. 循环搜索: 当节点  $root$  为空时跳出;

- 当  $p, q$  都在  $root$  的 右子树 中, 则遍历至  $root.right$ ;
- 否则, 当  $p, q$  都在  $root$  的 左子树 中, 则遍历至  $root.left$ ;
- 否则, 说明找到了 最近公共祖先, 跳出。

2. 返回值: 最近公共祖先  $root$ 。

**① 迭代**

```

TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    while(root != NULL){
        if(p->val < root->val && q->val < root->val){
            root = root->left;
        } else if(p->val > root->val && q->val > root->val){
            root = root->right;
        } else{
            return root;
        }
    }
    return root;
}

```

**② 递归**

```

TreeNode* ans = new TreeNode();
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    findparent(root, p, q);
    return ans;
}

void findparent(TreeNode* root, TreeNode* p, TreeNode* q){
    if(root == NULL) {return;}
    if(p->val < root->val && q->val < root->val){
        findparent(root->left, p, q);
    } else if(p->val > root->val && q->val > root->val){
        findparent(root->right, p, q);
    } else {ans = root;}
}

```

## 2. ① 哈希表

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    unordered_map<int, TreeNode*> parents;
    unordered_map<int, bool> visited;
    parents[root->val] = NULL;
    dfs(root, parents);
    while(p != NULL){
        visited[p->val] = true;
        p = parents[p->val];
    }
    while(q != NULL){
        if(visited[q->val] == true) {return q;}
        q = parents[q->val];
    }
    return NULL;
}

void dfs(TreeNode* root, unordered_map<int, TreeNode*>& parents){
    if(root->left != NULL){
        parents[root->left->val] = root;
        dfs(root->left, parents);
    }
    if(root->right != NULL){
        parents[root->right->val] = root;
        dfs(root->right, parents);
    }
}
```

## ② 深归

```
TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
    if(root == NULL) {return NULL;}
    if(root == p || root == q){
        return root; // 如果p和q中有等于root的，那么它们的最近公共祖先即为root（一个节点也可以是它自己的祖先）
    }
    // 递归遍历左/右子树，只要在左/右子树中找到了p或q，则先找到谁就返回谁
    TreeNode* left = lowestCommonAncestor(root->left, p, q);
    TreeNode* right = lowestCommonAncestor(root->right, p, q);
    if(left == NULL){
        return right; // 如果在左子树中p和q都找不到，则p和q一定都在右子树中，右子树中先遍历到的那个就是最近公共祖先
    }
    else if(right == NULL){
        return left; // 反之，如果left不为空，在左子树中找到节点（p或q），这时候要再判断一下右子树中的情况，如果在右子树中，p和q都找不到，则p和q一定都在左子树中，左子树中先遍历到的那个就是最近公共祖先
    }
    else {return root;} // 答案：当left和right均不为空时，说明p、q节点分别在root异侧，最近公共祖先即为 root
}
```

# 6. 字符串

## 字符串排序

2. 567. 字符串的排列：给定两个字符串 s1 和 s2，写一个函数来判断 s2 是否包含 s1 的排列。
3. 791. 自定义字符串排序：字符串 S 和 T 只包含小写字符。在 S 中，所有字符只会出现一次。S 已经根据某种规则进行了排序。我们要根据 S 中的字符顺序对 T 进行排序。更具体地说，如果 S 中 x 在 y 之前出现，那么返回的字符串中 x 也应出现在 y 之前。返回任意一种符合条件的字符串 T。
4. 953. 验证外星语词典：某种外星语也使用英文小写字母，但可能顺序 order 不同。字母表的顺序 (order) 是一些小写字母的排列。
5. 451. 根据字符出现频率排序：给定一个字符串，请将字符串里的字符按照出现的频率降序排列。
6. 937. 重新排列日志文件：按一定规则返回日志的最终顺序。

## 2.

输入: s1 = "ab" s2 = "eidbaooo"  
输出: true  
解释: s2 包含 s1 的排列之一 ("ba")。

使用两个数组  $cnt_1$  和  $cnt_2$ ,  $cnt_1$  统计  $s_1$  中各个字符的个数,  $cnt_2$  统计当前遍历的子串中各个字符的个数。

由于需要遍历的子串长度均为  $n$ , 我们可以使用一个固定长度为  $n$  的滑动窗口来维护  $cnt_2$ : 滑动窗口每向右滑动一次, 就多统计一次进入窗口的字符, 少统计一次离开窗口的字符。然后, 判断  $cnt_1$  是否与  $cnt_2$  相等, 若相等则意味着  $s_1$  的排列之一是  $s_2$  的子串。

## 滑动窗口法

```
bool checkInclusion(string s1, string s2) {
    int size1 = s1.size();
    int size2 = s2.size();
    if(size1 > size2) {return false;}
    vector<int> v1(26, 0), v2(26, 0);
    for(int i = 0; i < size1; i++){
        v1[s1[i] - 'a']++;
        v2[s2[i] - 'a']++;
    }
    if(v1 == v2) {return true;}
    for(int i = size1; i < size2; i++){
        v2[s2[i-size1] - 'a']--;
        v2[s2[i] - 'a']++;
        if(v1 == v2) {return true;}
    }
    return false;
}
```

## 3.

输入: order = "cba", s = "abcd"  
输出: "cbad"  
解释:  
“a”、“b”、“c”是按顺序出现的, 所以“a”、“b”、“c”的顺序应该是“c”、“b”、“a”。  
因为“d”不是按顺序出现的, 所以它可以在返回的字符串中的任何位置。  
“dcba”、“cdba”、“cbda”也是有效的输出。

## 先映射后快排

```
string customSortString(string order, string s) {
    vector<int> order_map(26, -1);
    for(int i = 0; i < order.size(); i++) {
        order_map[order[i] - 'a'] = i; //构建映射
    }
    quicksort(s, order_map, 0, (int)s.size() - 1);
    return s;
} ↳ 使用快排,根据字母排序
```

## 4.

某种外星语也使用英文小写字母, 但可能顺序 order 不同。字母表的顺序 (order) 是一些小写字母的排列。

给定一组用外星语书写的单词 words, 以及其字母表的顺序 order, 只有当给定的单词在这种外星语中按字典序排列时, 返回 true; 否则, 返回 false。

## 先映射后判断

```
bool isAlienSorted(vector<string>& words, string order) {
    vector<int> order_map(26);
    for(int i = 0; i < order.size(); i++) {
        order_map[order[i] - 'a'] = i;
    }
    for(int i = 1; i < words.size(); i++) {
        if(check(words[i-1], words[i], order_map) == false) {return false;} 先看相邻比较即可, 不用比较(x1>x2)+...+xi次
    }
    return true;
}

bool check(string& w1, string& w2, vector<int>& order_map) {
    for(int i = 0; i < w1.size(); i++) {
        if(i >= w2.size()) {return false;}
        if(w1[i] == w2[i]) {continue;} 相等
        if(order_map[w1[i] - 'a'] < order_map[w2[i] - 'a']) {return true;} 简化版
        else {return false;}
    }
    return true;
}
```

### 示例 1:

输入: words = ["hello", "leetcode"], order = "hlabcdefgijklnopqrstuvwxyz"  
输出: true  
解释: 在该语言的字母表中, 'h' 位于 'l' 之前, 所以单词序列是按字典序排列的。

5.

输入: s = "tree"  
 输出: "eert"  
 解释: 'e' 出现两次, 'r' 和 't' 都只出现一次。  
 因此 'e' 必须出现在 'r' 和 't' 之前。此外, "eetr" 也是一个有效的答案。

## 使用堆排序(大根堆,且用库实现)

```
string frequencySort(string s) {
    unordered_map<char, int> s_map;
    for(auto& c:s){
        s_map[c]++;
    }
    priority_queue<pair<int, char>, vector<pair<int, char>>, less<pair<int, char>>> s_bucket;
    for(auto& i:s_map){
        s_bucket.emplace(i.second, i.first);
    }
    string ans;
    while(!s_bucket.empty()){
        for(int i = 0; i < s_bucket.top().first; i++){
            ans += s_bucket.top().second;
        }
        s_bucket.pop();
    }
    return ans;
}
```

→ 大根堆 ⇒ pair类型的优先队列  
 first排序,且 first一样大  
 则看 second

也可用题3的方法解题

6.

给你一个日志数组 logs。每条日志都是以空格分隔的字符串，其第一个字符为字母与数字混合的 标识符。

有两种不同类型的日志：

- 字母日志：除标识符之外，所有字均由小写字母组成
- 数字日志：除标识符之外，所有字均由数字组成

请按如下规则将日志重新排序：

- 所有 字母日志 都排在 数字日志 之前。
- 字母日志 在内容不同时，忽略标识符后，按内容字母顺序排序；在内容相同时，按标识符排序。
- 数字日志 应该保留原来的相对顺序。

返回日志的最终顺序。

输入: logs = ["dig1 8 1 5 1", "let1 art can", "dig2 3 6", "let2 own kit dig", "let3 art zero"]  
 输出: ["let1 art can", "let3 art zero", "let2 own kit dig", "dig1 8 1 5 1", "dig2 3 6"]

解释:

字母日志的内容都不同，所以顺序为 "art can", "art zero", "own kit dig"。

数字日志保留原来的相对顺序 "dig1 8 1 5 1", "dig2 3 6"。

```
vector<string> reorderLogFiles(vector<string>& logs) {
    vector<pair<string, string>> chara_logs; // 存储字母日志
    vector<string> digit_logs; // 存储数字日志
    for(auto& w:logs){
        int pos = w.find_first_of(" "); // 找到第一个空格的位置
        if(isdigit(w[pos+1])){ // 判断是否为数字日志
            digit_logs.emplace_back(w);
        } else{
            string flag = w.substr(0, pos+1); // 标识符，从下标为0的字符开始提取 pos+1 个字符
            string content = w.substr(pos+1); // 内容，从下标为 pos+1 开始提取到最后
            chara_logs.emplace_back(flag, content);
        }
    }
    sort_chara(chara_logs, 0, (int)chara_logs.size()-1); // 快速排序
    vector<string> ans;
    for(auto& log:chara_logs){
        string tmp = log.first;
        tmp += log.second; // 连接回来
        ans.emplace_back(tmp);
    }
    for(auto& log:digit_logs){
        ans.emplace_back(log);
    }
    return ans;
}
```

```
int partition(vector<pair<string, string>>& chara_logs, int l, int r){
    int i = rand()%(r-l+1)+l;
    swap(chara_logs[i], chara_logs[r]);
    i = l-1;
    for(int j = l; j <= r-1; j++){
        if(chara_logs[j].second == chara_logs[r].second){ // 若内容相同
            if(chara_logs[j].first <= chara_logs[r].first){
                i++;
                swap(chara_logs[i], chara_logs[j]);
            }
        } else{
            if(chara_logs[j].second <= chara_logs[r].second){
                i++;
                swap(chara_logs[i], chara_logs[j]);
            }
        }
    }
    swap(chara_logs[i+1], chara_logs[r]);
    return i+1;
}
```

```
void sort_chara(vector<pair<string, string>>& chara_logs, int l, int r){
    if(l > r){
        int pos = partition(chara_logs, l, r);
        sort_chara(chara_logs, l, pos-1);
        sort_chara(chara_logs, pos+1, r);
    }
}
```

## 字符串相关操作

<https://blog.csdn.net/dengzqjia/article/details/119481443>

## 字符串匹配

1. 28. 实现 strStr(): 给定一个 haystack 字符串和一个 needle 字符串，在 haystack 字符串中找出 needle 字符串出现的第一个位置（从 0 开始）。如果不存在，则返回 -1。
2. 10. 正则表达式匹配和剑指 Offer 19. 正则表达式匹配：请实现一个函数用来匹配包含‘·’和‘\*’的正则表达式。模式中的字符‘·’表示任意一个字符，而‘\*’表示它前面的字符可以出现任意次（含 0 次）。在本题中，匹配是指字符串的所有字符匹配整个模式。例如，字符串“aaa”与模式“a.a”和“abaca”匹配，但与“aa.a”和“ab\*a”均不匹配。
3. 290. 单词规律：给定一种规律 pattern 和一个字符串 str，判断 str 是否遵循相同的规律。还有个 291. 单词规律2是会员的困难题，学有余力的会员可以试试看。
4. 面试题 17.17. 多次搜索：给定一个较长字符串 big 和一个包含较短字符串的数组 smalls，设计一个方法，根据 smalls 中的每一个较短字符串，对 big 进行搜索。输出 smalls 中的字符串在 big 里出现的所有位置 positions，其中 positions[i] 为 smalls[i] 出现的所有位置。
5. 面试题 16.18. 模式匹配：你有两个字符串，即 pattern 和 value。pattern 字符串由字母“a”和“b”组成，用于描述字符串中的模式。例如，字符串“catcatgocatgo”匹配模式“aabab”（其中“cat”是“a”，“go”是“b”），该字符串也匹配像“a”、“ab”和“b”这样的模式。但需注意“a”和“b”不能同时表示相同的字符串。编写一个方法判断 value 字符串是否匹配 pattern 字符串。
6. 686. 重复叠加字符串匹配：给定两个字符串 a 和 b，寻找重复叠加字符串 a 的最小次数，使得字符串 b 成为叠加后的字符串 a 的子串，如果不存在则返回 -1。

## 1. ① 滑动窗口（暴力解法）

```
int strStr(string haystack, string needle) {  
    int window_size = needle.size();  
    if(window_size == 0) {return 0;}  
    int word_size = haystack.size();  
    for(int i = 0; i<=word_size-window_size; i++){  
        if(check(haystack, needle, i, window_size)) {return i;}  
    }  
    return -1;  
}  
  
bool check(string& haystack, string& needle, int i, int window_size){  
    for(int m = 0; m<window_size; m++){  
        if(haystack[i+m] != needle[m]) {return false;}  
    }  
    return true;  
}
```

## ② KMP 算法

```
int strStr(string haystack, string needle) {  
    if(needle.size() == 0) {return 0;} 这句“叠加，且是后缀  
    vector<int> next(needle.size());  
    for(int left = 0, right = 1; right < needle.size(); right++){  
        while(left > 0 && needle[right] != needle[left]) {  
            left = next[left-1];  
        }  
        if(needle[right] == needle[left]) {  
            left++;  
        }  
        next[right] = left;  
    }  
    for(int j = 0, i = 0; i < haystack.size(); i++){  
        while(j > 0 && haystack[i] != needle[j]) {  
            j = next[j-1];  
        }  
        if(haystack[i] == needle[j]) {  
            j++;  
        }  
        if(j == needle.size()) {return i - needle.size() + 1;}  
    }  
    return -1;  
}
```

<https://leetcode.cn/problems/implement-strstr/solution/duo-tu-yu-jing-xiang-jie-kmp-suan-fa-by-w3c9c/>

<https://leetcode.cn/problems/implement-strstr/solution/shua-chuan-lc-shuang-bai-po-su-jie-fa-km-tb86/>

next 数组存放的是当前长度下的最长相同前后缀的长度

## 2.

给你一个字符串 s 和一个字符规律 p，请你来实现一个支持 ‘.’ 和 ‘\*’ 的正则表达式匹配。

- ‘.’ 匹配任意单个字符
- ‘\*’ 匹配零个或多个前面的那个元素

所谓匹配，是要涵盖 整个 字符串 s 的，而不是部分字符串。

## 使用动态规划（不理解=-=）

<https://leetcode.cn/problems/regular-expression-matching/solution/shou-hui-tu-jie-wo-tai-nan-liao-by-hyj8/>

```

bool isMatch(string s, string p) {
    int s_size = s.size(), p_size = p.size();
    vector<vector<bool>> dp(s_size+1, vector<bool>(p_size+1, false)); //因为包含了空字符串
    //初始化, dp[i][j] 表示 s[:i] 与 p[:j] 是否匹配, 各自前 i, j 个是否匹配
    dp[0][0] = true; //两个空字符串
    for(int j = 1; j < p_size+1; j++){//找出s为空, p因为*为为空的情况
        if(p[j-1] == '*') {dp[0][j] = dp[0][j-2];}
    }
    //更新
    for(int i = 1; i < s_size+1; i++){
        for(int j = 1; j < p_size+1; j++){
            //情况1: 符合, 直接更新
            if(s[i-1] == p[j-1] || p[j-1] == '.'){
                dp[i][j] = dp[i-1][j-1];
            }
            //情况2: 考虑*的情况
            else if(p[j-1] == '*'){
                if(s[i-1] == p[j-2] || p[j-2] == '.'){
                    dp[i][j] = dp[i][j-2] || dp[i-1][j-2] || dp[i-1][j]; //分别是重复0次、重复一次、重复两次及以上
                }
                else{///s[i-1]与p[j-2]不匹配, *需要重复0次
                    dp[i][j] = dp[i][j-2];
                }
            }
        }
    }
    return dp[s_size][p_size];
}

```

### 3.

示例1:

输入: pattern = "abba", s = "dog cat cat dog"  
输出: true

示例 2:

输入: pattern = "abba", s = "dog cat cat fish"  
输出: false

### 使用两个哈希表,才能实现两键唯一对应

```

bool wordPattern(string pattern, string s) {
    if(pattern.size() == 0 || s.size() == 0) {return false;}
    unordered_map<int, string> pattern_map;
    unordered_map<string, int> words_map;
    s += " ";
    for(int i = 0; i < pattern.size(); i++){
        if(s == "") {return false;} // pattern与s大小不等,如"aaaa", "dog dog dog"
        int pos = s.find_first_of(" ");
        string word = s.substr(0, pos);
        if(pattern_map.count(pattern[i]-'a') && pattern_map[pattern[i]-'a'] != word){
            return false;
        }
        if(words_map.count(word) && words_map[word] != pattern[i]-'a'){
            return false;
        }
        pattern_map[pattern[i]-'a'] = word;
        words_map[word] = pattern[i]-'a';
        s = s.substr(pos+1);
    }
    if(s != "") {return false;} // pattern与s大小不等,如"aaa", "dog dog dog"
    return true;
}

```

### 4.

给定一个较长字符串 big 和一个包含较短字符串的数组 smalls，设计一个方法，根据 smalls 中的每一个较短字符串，对 big 进行搜索。输出 smalls 中的字符串在 big 里出现的所有位置 positions，其中 positions[i] 为 smalls[i] 出现的所有位置。

示例:

输入:  
big = "mississippi"  
smalls = ["is", "ppi", "hi", "sis", "i", "ssippi"]  
输出: [[1,4],[8,[]],[3,[1,4,7,10],[5]]]

### ① KMP 算法

```

vector<vector<int>> multiSearch(string big, vector<string>& smalls) {
    vector<vector<int>> ans(smalls.size());
    for(int i = 0; i < smalls.size(); i++){
        kmp(i, smalls[i], big, ans);
    }
    return ans;
}

```

## ② 前缀树

```
struct TrieNode{  
    int index; //新增加一个成员，用于记录此时字符串在small中的索引  
    string word;  
    unordered_map<char, TrieNode*> children;  
    TrieNode(){  
        this->index = -1;  
        this->word = "";  
    }  
};  
  
void insertTrie(TrieNode* root, int i, string& word){  
    TrieNode* node = root;  
    for(auto& c:word){  
        if(!node->children.count(c)){  
            node->children[c] = new TrieNode();  
        }  
        node = node->children[c];  
    }  
    node->index = i;  
    node->word = word;  
}
```

```
void search(int i, string& big, TrieNode* root, vector<vector<int>>& ans){  
    for(int j = i; j<big.size(); j++){  
        if(!root->children.count(big[j])) {return;}  
        root = root->children[big[j]];  
        if(root->index > -1){ //找一个匹配串  
            ans[root->index].emplace_back(i);  
        }  
    }  
    return;  
}  
  
vector<vector<int>> multiSearch(string big, vector<string>& smalls) {  
    TrieNode* root = new TrieNode();  
    for(int i = 0; i<smalls.size(); i++){  
        insertTrie(root, i, smalls[i]);  
    }  
    vector<vector<int>> ans(smalls.size());  
    for(int i = 0; i<big.size(); i++){  
        search(i, big, root, ans);  
    }  
    return ans;  
}
```

## 5. 太复杂了，难以理解 =\_=

## 6.

输入: a = "abcd", b = "cdabcdab"

输出: 3

解释: a 重复叠加三遍后为 "abcdabcdabcd"，此时 b 是其子串。

在这个代码  
下是可以  
找到

```
int repeatedStringMatch(string a, string b) {  
    string a_copy = "";  
    while(a_copy.size() <= b.size()){  
        a_copy += a;  
    }  
    a_copy += a;  
    int index = kmp(a_copy, b);  $\Rightarrow$  KMP 算法  
    if(index == -1) {return -1;}  
    return (index + b.size() - 1)/a.size() + 1;  
}
```

## 异位词

242. 有效的字母异位词和面试题 01.02. 判定是否互为字符重排：给定两个字符串 s1 和 s2，请编写一个程序，确定其中一个字符串的字符重新排列后，能否变成另一个字符串。
49. 字母异位词分组和面试题 10.02. 变位词组：对字符串数组进行排序，将所有变位词组合在一起。变位词是指字母相同，但排列不同的字符串。
438. 找到字符串中所有字母异位词：给定一个字符串 s 和一个非空字符串 p，找到 s 中所有是 p 的字母异位词的子串，返回这些子串的起始索引。

1. 给定两个字符串  $s$  和  $t$ ，编写一个函数来判断  $t$  是否是  $s$  的字母异位词。

注意：若  $s$  和  $t$  中每个字符出现的次数都相同，则称  $s$  和  $t$  互为字母异位词。

## ① 哈希表

```
bool isAnagram(string s, string t) {
    if(s.size() != t.size()) {return false;}
    unordered_map<char, int> c_map;
    for(auto& c:s){
        if(c_map.count(c)) {c_map[c] = 1;}
        else {c_map[c]++;}
    }
    for(auto& c:t){
        if(!c_map.count(c)) {return false;}
        else {c_map[c]--;}
    }
    for(auto& c:c_map){
        if(c.second != 0) {return false;}
    }
    return true;
}
```

## 2.

给你一个字符串数组，请你将字母异位词组合在一起。可以按任意顺序返回结果列表。

字母异位词 是由重新排列源单词的字母得到的一个新单词，所有源单词中的字母通常恰好只用一次。

示例 1：

```
输入: strs = ["eat", "tea", "tan", "ate", "nat", "bat"]
输出: [["bat"], ["nat", "tan"], ["ate", "eat", "tea"]]
```

## ② 排序后比较

```
bool isAnagram(string s, string t) {
    quicksort(s, 0, s.size()-1);
    quicksort(t, 0, t.size()-1);
    if(s == t) {return true;}
    else {return false;}
}
```

## 排序(也可以用哈希表方法)

```
vector<vector<string>> groupAnagrams(vector<string>& strs) {
    unordered_map<string, vector<string>> strs_map;
    for(auto& w:strs){
        string s_tmp = w;
        quicksort(s_tmp, 0, (int)w.size()-1);
        strs_map[s_tmp].emplace_back(w);
    }
    vector<vector<string>> ans;
    for(auto& ws:strs_map){
        ans.emplace_back(ws.second);
    }
    return ans;
}
```

## 3.

输入：  $s = "cbaebabacd"$ ,  $p = "abc"$

输出：  $[0,6]$

解释：

起始索引等于 0 的子串是 "cba"，它是 "abc" 的异位词。

起始索引等于 6 的子串是 "bac"，它是 "abc" 的异位词。

## ① 滑动窗口

```
vector<int> findAnagrams(string s, string p) {
    vector<int> ans;
    vector<int> s_map(26, 0);
    vector<int> p_map(26, 0);
    int s_size = s.size(), p_size = p.size();
    if(s_size < p_size) {return ans;}
    for(int i = 0; i < p_size; i++){
        s_map[s[i]-'a']++;
        p_map[p[i]-'a']++;
    }
    if(s_map == p_map) {ans.emplace_back(0);}
    for(int i = p_size; i < s_size; i++){
        s_map[s[i-p_size]-'a']--;
        s_map[s[i]-'a']++;
        if(s_map == p_map) {ans.emplace_back(i-p_size+1);}
    }
    return ans;
}
```

## ②优化滑动窗口

在方法一的基础上，我们不再分别统计滑动窗口和字符串  $p$  中每种字母的数量，而是统计滑动窗口和字符串  $p$  中每种字母数量的差；并引入变量  $differ$  来记录当前窗口与字符串  $p$  中数量不同的字母的个数，并在滑动窗口的过程中维护它。

在判断滑动窗口中每种字母的数量与字符串  $p$  中每种字母的数量是否相同时，只需要判断  $differ$  是否为零即可。

```
vector<int> findAnagrams(string s, string p) {
    vector<int> ans;
    int s_size = s.size(), p_size = p.size();
    if(s_size < p_size) {return ans;}
    vector<int> count(26, 0);
    for(int i = 0; i < p_size; i++){
        count[s[i] - 'a']++;
        count[p[i] - 'a']--;
    }
    int differ = 0;
    for(auto& c:count){
        if(c != 0) {differ++;}
    }
    if(differ == 0) {ans.emplace_back(0);}
    for(int i = p_size; i < s_size; i++){
        if(count[s[i-p_size] - 'a'] == 1) {differ--;} // 窗口中字母 s[i-p_size] 的数量与字符串 p 中的数量从不同变得相同
        else if(count[s[i-p_size] - 'a'] == 0) {differ++;} // 窗口中字母 s[i-p_size] 的数量与字符串 p 中的数量从相同变得不同
        count[s[i-p_size] - 'a']--;
        if(count[s[i] - 'a'] == -1) {differ--;} // 窗口中字母 s[i] 的数量与字符串 p 中的数量从不同变得相同
        else if(count[s[i] - 'a'] == 0) {differ++;} // 窗口中字母 s[i] 的数量与字符串 p 中的数量从相同变得不同
        count[s[i] - 'a']++;
        if(differ == 0) {ans.emplace_back(i-p_size+1);}
    }
    return ans;
}
```

## 字符串压缩和解码

- 面试题 01.06. 字符串压缩：**字符串压缩。利用字符重复出现的次数，编写一种方法，实现基本的字符串压缩功能。比如，字符串aabcccccaa会变为a2b1c5a3。若“压缩”后的字符串没有变短，则返回原先的字符串。你可以假设字符串中只包含大小写英文字母（a至z）。
- 443. 压缩字符串：**给定一组字符，使用原地算法将其压缩。压缩后的长度必须始终小于或等于原数组长度。数组的每个元素应该是长度为1的字符（不是 int 整数类型）。在完成原地修改输入数组后，返回数组的新长度。
- 394. 字符串解码：**给定一个经过编码的字符串，返回它解码后的字符串。编码规则为： $k[encoded\_string]$ ，表示其中方括号内部的  $encoded\_string$  正好重复  $k$  次。注意  $k$  保证为正整数。

1.

```
string compressString(string s) {
    string s_compress = "";
    char t = s[0];
    int t_cnt = 0;
    for(auto& c:s){
        if(c != t){
            s_compress += t; s_compress += to_string(t_cnt);
            t = c; t_cnt = 0;
        }
        t_cnt++;
    }
    s_compress += t; s_compress += to_string(t_cnt);
    if(s_compress.size() < s.size()) {return s_compress;}
    else {return s;}
}
```

2.

给你一个字符数组  $\text{chars}$ ，请使用下述算法压缩：

从一个空字符串  $s$  开始。对于  $\text{chars}$  中的每组 连续重复字符：

- 如果这一组长度为 1，则将字符追加到  $s$  中。
- 否则，需要向  $s$  追加字符，后跟这一组的长度。

压缩后得到的字符串  $s$  不应该直接返回，需要转储到字符数组  $\text{chars}$  中。需要注意的是，如果组长度为 10 或 10 以上，则在  $\text{chars}$  数组中会被拆分为多个字符。

请在 修改完输入数组后，返回该数组的新长度。

输入:  $\text{chars} = ["a", "a", "b", "b", "c", "c", "c"]$   
输出: 返回 6，输出数组的前 6 个字符应该是：  
["a", "2", "b", "2", "c", "3"]

解释: "aa" 被 "a2" 替代, "bb" 被 "b2" 替代, "ccc" 被 "c3" 替代。

```
int compress(vector<char>& chars) {
    string s = "";
    char m = chars[0];
    int m_cnt = 0;
    for(auto& c:chars){
        if(c != m){
            s += m;
            if(m_cnt > 1) {s += to_string(m_cnt);}
            m = c; m_cnt = 0;
        }
        m_cnt++;
    }
    s += m;
    if(m_cnt > 1) {s += to_string(m_cnt);}
    chars.resize(s.size());
    for(auto& c:s){
        chars.emplace_back(c);
    }
    return chars.size();
}
```

### 3.

示例 1:

输入: s = "3[a]2[bc]"  
输出: "aaabcbc"

示例 2:

输入: s = "3[a2[c]]"  
输出: "accaccacc"

示例 3:

输入: s = "2[abc]3[cd]ef"  
输出: "abcabccddcdcddef"

示例 4:

输入: s = "abc3[cd]xyz"  
输出: "abccdcdcdxyz"

```

string decodeString(string s) {
    string ans = "";
    stack<int> num_stack;
    stack<string> s_stack;
    int num = 0;
    for(auto& c:s){
        if(c == '['){
            num_stack.push(num);
            s_stack.push(ans);
            num = 0; ans = "";
        }
        else if(c == ']'){
            string tmp = "";
            int thisnum = num_stack.top();
            num_stack.pop();
            for(int i = 0; i<thisnum; i++) {tmp += ans;}
            ans = s_stack.top() + tmp;
            s_stack.pop();
        }
        else if(isdigit(c)) {num = num*10 + (c-'0');}
        else {ans += c;}
    }
    return ans;
}

```

## 字符串反转、翻转和旋转

1. 344. 反转字符串: 其作用是将输入的字符串反转过来。输入字符串以字符数组 char[] 的形式给出。不要给另外的数组分配额外的空间，你必须原地修改输入数组、使用 O(1) 的额外空间解决这一问题。

2. 541. 反转字符串 II: 给定一个字符串 s 和一个整数 k, 你需要对从字符串开头算起的每隔 2k 个字符的前 k 个字符进行反转。

3. 剑指 Offer 58 - II. 左旋转字符串: 字符串的左旋转操作是把字符串前面的若干个字符转移到字符串的尾部。请定义一个函数实现字符串左旋转操作的功能。比如，输入字符串"abcdefg"和数字2，该函数将返回左旋转两位得到的结果"cdefgab"。

4. 796. 旋转字符串和面试题 01.09. 字符串轮转: 给定两个字符串s1和s2, 请编写代码检查s2是否为s1旋转而成。

5. 151. 翻转字符串里的单词和剑指 Offer 58 - I. 翻转单词顺序: 输入一个英文句子，翻转句子中单词的顺序，但单词内字符的顺序不变。为简单起见，标点符号和普通字母一样处理。例如输入字符串"i am a student .", 则输出"student . a am I"。

**提升1:** 尝试让结尾依旧是" ."，且结尾的" ."与前一个单词之间无空格。比如：输入一个英文句子，词之间有1个或者若干个空格，句子以英文标点" ."结尾。要求颠倒该句子中的词语顺序，并且词之间有且只有一个空格，结尾仍然是" ."，结尾的" ."与前一个单词之间无空格。

**提升2:** 如果输入是字符数组，如何在不分配额外空间的情况下，得到和上述一样的效果。

6. 557. 反转字符串中的单词 III: 给定一个字符串，你需要反转字符串中每个单词的字符顺序，同时仍保留空格和单词的初始顺序。

→可执行去除空格函数后把'.'删掉，最后的时候再补上

## 1. 双指针

```

void reverseString(vector<char>& s) {
    int s_size = s.size();
    for(int left = 0, right = s_size-1; left<right; left++, right--){
        swap(s[left], s[right]);
    }
}

```

## 2. 双指针

给定一个字符串 s 和一个整数 k, 从字符串开头算起，每计数至 2k 个字符，就反转这 2k 字符中的前 k 个字符。

- 如果剩余字符少于 k 个，则将剩余字符全部反转。
- 如果剩余字符小于 2k 但大于或等于 k 个，则反转前 k 个字符，其余字符保持原样。

```

string reverseStr(string s, int k) {
    int s_size = s.size();
    for(int left = 0; left<s_size; left = left+2*k){
        int right = left+k-1;
        restr(left, min(right, s_size-1), s);
    }
    return s;
}

```

山题的函数

3. string reverseLeftWords(string s, int n) {  
 string ans = s.substr(n);  
 ans += s.substr(0, n);  
 return ans;  
}

## 5. ① 空间复杂度大

```
string reverseWords(string s) {  

    string tmp = "";  

    vector<string> s_map;  

    for(auto& c:s){  

        if(c == ' '){  

            if(tmp.size() != 0) {  

                s_map.emplace_back(tmp);  

                tmp = "";  

            }  

        }  

        else{  

            tmp += c;  

        }  

    }  

    if(tmp.size() != 0) {  

        s_map.emplace_back(tmp);  

        tmp = "";  

    }  

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

        tmp += s_map[i]; tmp += ' ';  

    }  

    tmp.pop_back();  

    return tmp;  

}
```

4. 给定两个字符串 s 和 goal。如果在若干次旋转操作之后， s 能变成 goal，那么返回 true。

s 的 旋转操作 就是将 s 最左边的字符移动到最右边。

- 例如：若 s = 'abcde'，在旋转一次之后结果就是 'bcdea'。

首先，如果 s 和 goal 的长度不一样，那么无论怎么旋转，s 都不能得到 goal，返回 false。字符串 s+s 包含了所有 s 通过旋转操作得到的字符串，只需要检查 goal 是否为 s+s 的子字符串即可。

```
bool rotateString(string s, string goal) {  

    if(s.size() != goal.size()) {return false;}  

    s += s;  

    return kmp(goal, s);  

}
```

## 空间复杂度 O(1)

```
string reverseWords(string s) {  

    removeExtraSpace(s);  

    reverse(s, 0, s.size()-1);  

    int start = 0;  

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

        if(i == s.size() || s[i] == ' '){  

            reverse(s, start, i-1);  

            start = i+1;  

        }  

    }  

    return s;  

}
```

## 原地去除空格

```
void removeExtraSpace(string& s){  

    int slow = 0, fast = 0;//快慢指针  

    //去掉字符串前面的空格  

    while(s.size() > 0 && fast < s.size() && s[fast] == ' ') {fast++;}  

    //去掉单词间的多余空格，只留下一个  

    for(;fast < s.size(); fast++){  

        if(fast > 1 && s[fast-1] == s[fast] && s[fast] == ' ') {continue;}  

        else{  

            s[slow] = s[fast];  

            slow++;  

        }  

    }  

    //去掉字符串末尾的空格  

    if(slow > 1 && s[slow-1] == ' ') {s.resize(slow-1);}  

    else {s.resize(slow);}  

}
```

## 翻转字符串指定区域

```
void reverse(string& s, int start, int end){  

    for(int left = start, right = end; left<right; left++, right--){  

        swap(s[left], s[right]);  

    }  

}
```

<https://leetcode.cn/problems/reverse-words-in-a-string/solution/by-carlsun-2-r8jt/>

## 6. 与题5的主函数相似

- 移除多余空格
- 将整个字符串反转
- 将每个单词反转

## 单词

可以直接按关键词【单词】搜索，以下仅列举一些比较常见的。

1. **720.词典中最长的单词**: 给出一个字符串数组words组成的一本英语词典。从中找出最长的一个单词，该单词是由words词典中其他单词逐步添加一个字母组成。若其中有多个可行的答案，则返回答案中字典序最小的单词。
2. **819.最常见的单词**: 给定一个段落(paragraph)和一个禁用单词列表(banned)。返回出现次数最多，同时不在禁用列表中的单词。
3. **面试题 17.15.最长单词**: 给定一组单词words，编写一个程序，找出其中的最长单词，且该单词由这组单词中的其他单词组合而成。若多个长度相同的结果，返回其中字典序最小的一项，若没有符合要求的单词则返回空字符串。
4. **面试题 17.22.单词转换**: 给定字典中的两个词，长度相等。写一个方法，把一个词转换成另一个词，但是一次只能改变一个字符。每一步得到的新词都必须能在字典中找到。编写一个程序，返回一个可能的转换序列。如有多个可能的转换序列，你可以返回任何一个。
5. **面试题 17.11.单词距离**和会员题**245.最短单词距离3**: 有个内含单词的超大文本文档，给定任意两个单词，找出在这个文件中这两个单词的最短距离(相隔单词数)。



输入: words = ["w","wo","wor","worl", "world"]  
输出: "world"  
解释: 单词"world"可由"w", "wo", "wor", 和 "worl"逐步添加一个字母组成。

字典树 + DFS 遍历  
字典树

2.

```
string mostCommonWord(string paragraph, vector<string>& banned) {  
    unordered_map<string, int> w_map;  
    string tmp = "";  
    string ans = "";  
    int ans_cnt = 0;  
    for(int i = 0; i < paragraph.size(); i++){  
        if(isalpha(paragraph[i])){ // 判断是否为字母  
            if(tmp.size() > 0){  
                w_map[tmp]++;  
                tmp = "";  
            }  
        }  
        else{  
            if(paragraph[i] < 'a') {paragraph[i] = paragraph[i] + 'a' - 'A';}  
            tmp += paragraph[i];  
        }  
    }  
    if(tmp.size() > 0) {w_map[tmp]++;}  
    for(auto& w:w_map){  
        if(!find(banned.begin(), banned.end(), w.first) != banned.end()) {continue;}  
        if(w.second > ans_cnt){  
            ans = w.first;  
            ans_cnt = w.second;  
        }  
    }  
    return ans;  
}
```

```
string longestWord(vector<string>& words) {  
    TrieNode* root = new TrieNode();  
    for(int i = 0; i < words.size(); i++){  
        insertTrieNode(root, words[i], i);  
    }  
    string ans = "";  
    int ans_index = -1;  
    for(auto& child:root->children){  
        dfs(child.second, ans, ans_index);  
    }  
    return ans;  
}  
  
void dfs(TrieNode* root, string& ans, int& ans_index){  
    if(root->index == -1) {return;}  
    else{  
        if(root->word.size() > ans.size()) {  
            ans = root->word;  
            ans_index = root->index;  
        }  
        else if(root->word.size() == ans.size() && root->word < ans){  
            ans = root->word;  
            ans_index = root->index;  
        }  
    }  
    for(auto& child:root->children){  
        dfs(child.second, ans, ans_index);  
    }  
}
```



1.

输入:  
["cat", "banana", "dog", "nana", "walk", "walker", "dogwalker"]  
输出: "dogwalker"  
解释: "dogwalker"可由"dog"和"walker"组成。

## 字典树十回溯

```
string longestWord(vector<string>& words) {
    TrieNode* root = new TrieNode();
    for(auto& w:words){
        insert(root, w);
    }
    string ans = "";
    for(auto& w:words){
        char c = w[0];
        if(w.size() < ans.size()) {continue;}
        if(w.size() == ans.size() && w > ans) {continue;}
        if(dfs(root, w, 1)) {ans = w;}
    }
    return ans;
}

bool dfs(TrieNode* root, string& word, int flag){
    if(word.size() == 0) {return true;}
    TrieNode* node = root;
    for(int i = 0; i<word.size()-flag; i++){
        TrieNode* tmp = node->children[word[i]];
        if(tmp == nullptr) {return false;}
        if(tmp->word.size() > 0){
            string s_sub = word.substr(i+1);
            if(dfs(tmp, s_sub, 0)) {return true;}
        }
        node = node->children[word[i]];
    }
    return false;
}
```

4.

输入:  
beginWord = "hit",  
endWord = "cog",  
wordList = ["hot", "dot", "dog", "lot", "log", "cog"]  
  
输出:  
["hit", "hot", "dot", "lot", "log", "cog"]

## 回溯

```
vector<string> findLadders(string beginWord, string endWord, vector<string>& wordList) {
    vector<bool> visited(wordList.size(), false);
    vector<string> ans;
    ans.emplace_back(beginWord);
    if(dfs(beginWord, endWord, visited, wordList, ans)) {return ans;}
    else {ans.resize(0); return ans;}
}

bool dfs(string& curWord, string& endWord, vector<bool>& visited, vector<string>& wordList, vector<string>& ans) {
    if(curWord == endWord) {return true;}
    for(int i = 0; i<wordList.size(); i++){
        if(visited[i] || !check(curWord, wordList[i])) {continue;}
        visited[i] = true;
        ans.emplace_back(wordList[i]);
        if(dfs(wordList[i], endWord, visited, wordList, ans)) {return true;}
        ans.pop_back();
        //这里不用加 visited[i] = false; (剪枝)
    }
    return false;
}

bool check(string& curWord, string& wordListi){
    if(curWord.size() != wordListi.size()) {return false;}
    int cnt = 0;
    for(int i = 0; i<curWord.size(); i++){
        if(curWord[i] != wordListi[i]) {cnt++;}
    }
    return cnt == 1;
}
```

5.

```
int findClosest(vector<string>& words, string word1, string word2) {
    int p = -1, q = -1, cnt = INT_MAX;
    for(int i = 0; i<words.size(); i++){
        if(words[i] == word1) {p = i;}
        if(words[i] == word2) {q = i;}
        if(p != -1 && q != -1) {cnt = min(cnt, abs(p-q));}
    }
    return cnt;
}
```

## 替换字符串

- 剑指 Offer 05. 替换空格：请实现一个函数，把字符串 s 中的每个空格替换成 "%20"。
- 面试题 01.03. URL化：URL化。编写一种方法，将字符串中的空格全部替换成 "%20"。假定该字符串尾部有足够的空间存放新增字符，并且知道字符串的“真实”长度。（注：用 Java 实现的话，请使用字符数组实现，以便直接在数组上操作。）
424. 替换后的最长重复字符：给你一个仅由大写英文字母组成的字符串，你可以将任意位置上的字符替换成另外的字符，总共可最多替换 k 次。在执行上述操作后，找到包含重复字母的最长子串的长度。

## 1.2.



```
string replaceSpace(string s) {
    string ans = "";
    for(int i = 0; i < s.size(); i++){
        if(s[i] == ' ') {ans += "%20";}
        else {ans += s[i];}
    }
    return ans;
}
```

输入: s = "AABABBA", k = 1

输出: 4

解释:

将中间的一个'A'替换成'B'，字符串变为 "AABBBBA"。  
子串 "BBB" 有最长重复字母，答案为 4。

## 使用滑动窗口

```
int characterReplacement(string s, int k) {
    vector<int> s_cnt(26, 0);
    // maxcnt记录字符出现次数最多那个字符的次数，size储存最大的窗口大小
    int left = 0, size = 0, maxcnt = INT_MIN;
    for(int i = 0; i < s.size(); i++){
        s_cnt[s[i] - 'A']++;
        // 比较之前记录的最大数 和 当前字符的数量
        maxcnt = max(maxcnt, s_cnt[s[i] - 'A']);
        // 若当前窗口大小 减去 窗口中最多相同字符的个数，大于 k 时
        while(i - left + 1 - maxcnt > k){
            s_cnt[s[left] - 'A']--;
            left++; // 滑动窗口的最左边向右挪动一位
        }
        size = max(size, i - left + 1);
    }
    return size;
}
```

## 子串

可以直接按关键词【子串】搜索，以下仅列举一些比较常见的。

696. 计数二进制子串：给定一个字符串 s，计算具有相同数量 0 和 1 的非空（连续）子字符串的数量，并且这些子字符串中的所有 0 和所有 1 都是组合在一起的。
459. 重复的子字符串：给定一个非空的字符串，判断它是否可以由它的一个子串重复多次构成。给定的字符串只含有小写英文字母，并且长度不超过 10000。
3. 无重复字符的最长子串 和 剑指 Offer 48. 最长不含重复字符的子字符串：给定一个字符串，请你找出其中不含有重复字符的最长子串的长度。
395. 至少有K个重复字符的最长子串：找到给定字符串（由小写字符组成）中的最长子串 T，要求 T 中的每一字符出现次数都不少于 k。输出 T 的长度。
1044. 最长重复子串：给出一个字符串 S，考虑其所有重复子串（S 的连续子串，出现两次或多次，可能会有重叠）。返回任何具有最长可能长度的重复子串。（如果 S 不含重复子串，那么答案为 ""。）PS：本题较难，量力而行。

输入: s = "00110011"  
输出: 6  
解释: 6 个子串满足具有相同数量的连续 1 和 0  
: "0011"、"01"、"1100"、"10"、"0011" 和 "01"。  
注意，一些重复出现的子串（不同位置）要统计它们出现的次数。  
另外，“00110011”不是有效的子串，因为所有的 0（还有 1）没有组合在一起。

我们可以将字符串 s 按照 0 和 1 的连续段分组，存在 counts 数组中，例如 s = 00110011，可以得到这样的 counts 数组：counts = [2, 3, 1, 2]。

这里 counts 数组中两个相邻的数一定代表的是两种不同的字符。假设 counts 数组中两个相邻的数字为 u 或者 v，它们对应着 u 个 0 和 v 个 1，或者 u 个 1 和 v 个 0。它们能组成的满足条件的子串数目为  $\min(u, v)$ ，即一对相邻的数字对答案的贡献。

我们只要遍历所有相邻的数对，求它们的贡献总和，即可得到答案。

```
int countBinarySubstrings(string s) {
    vector<int> cnts;
    int cur = 0;
    while(cur < s.size()){
        char c = s[cur];
        int cnt = 0;
        while(s[cur] == c && cur < s.size()){
            cnt++; cur++;
        }
        cnts.emplace_back(cnt);
    }
    int ans = 0;
    for(int i = 1; i < cnts.size(); i++){
        ans += min(cnts[i-1], cnts[i]);
    }
    return ans;
}
```

## 2. ① 暴力解法

```
bool repeatedSubstringPattern(string s) {
    int size = s.size();
    string tmp = "";
    for(int i = 0; i<size/2; i++){
        tmp += s[i];
        string tmp2 = tmp;
        while(tmp2.size() < s.size()) {tmp2 += tmp;}
        if(tmp2.size() > s.size()) {continue;}
        if(tmp2 == s) {return true;}
    }
    return false;
}
```

## KMP

假设字符串s使用多个重复子串构成(这个子串是最小重复单位), 重复出现的子字符串长度是x, 所以s是由n \* x组成。

因为字符串s的最长相同前后缀的长度一定是不包含s本身, 所以最长相同前后缀长度必然是m \* x, 而且n - m = 1, (这里如果不懂, 看上面的推论)

所以如果  $n \times (n - m)x = 0$ , 就可以判定有重复出现的子字符串。

数组长度为: len,

如果  $len \% (n - m)x = 0$ , 则说明(数组长度-最长相等前后缀的长度)正好可以被数组的长度整除, 说明有该字符串有重复的子字符串。

数组长度减去最长相等前后缀的长度相当于是第一个周期的长度, 也就是一个周期的长度, 如果这个周期可以被整除, 就说明整个数组就是这个周期的循环。

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

void kmp_cut(string& s, vector<int>& next){
    for(int left = 0, right = 1; right<s.size(); right++){
        while(left > 0 && s[left] != s[right]){
            left = next[left-1];
        }
        if(s[left] == s[right]) {left++;}
        next[right] = left;
    }
}
```

<https://leetcode.cn/problems/repeated-substring-pattern/solution/by-carlsun-2-g3iz/>



## 滑动窗口 + set

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



## 分治(DFS)

对于字符串s, 如果存在某个字符ch, 它的出现次数大于0且小于k, 则任何包含ch的子串都不可能满足要求。也就是说, 我们将字符串按照ch切分成若干段, 则满足要求的最长子串一定出现在某个被切分的段内, 而不能跨越一个或多个段。因此, 可以考虑分治的方式求解本题。

```
int longestSubstring(string s, int k) {
    return dfs(s, 0, s.size()-1, k);
}
```

```
int dfs(string& s, int l, int r, int k){
    vector<int> cnt(26, 0);
    for(int i = l; i <= r; i++){
        cnt[s[i]-'a']++;
    }
    char split = 0;
    for(int i = 0; i < 26; i++){
        if(cnt[i] > 0 && cnt[i] < k){
            split = i+'a';
            break;
        }
    }
    if(split == 0) {return r-l+1;}
    int i = l;
    int ret = 0;
    while(i <= r){
        while(i<r && s[i]==split) {i++;}
        if(i > r) {break;}
        int start = i;
        while(i<r && s[i]!=split) {i++;}
        int end = i-1;
        int length = dfs(s, start, end, k);
        ret = max(ret, length);
    }
    return ret;
}
```

# 5. 太难了 =\_=

## 唯一字符

- 剑指 Offer 50. 第一个只出现一次的字符：在字符串 s 中找出第一个只出现一次的字符。如果没有，返回一个单空格。s 只包含小写字母。
387. 字符串中的第一个唯一字符：给定一个字符串，找到它的第一个不重复的字符，并返回它的索引。如果不存在，则返回 -1。
- 面试题 01.01. 判定字符是否唯一：确定一个字符串 s 的所有字符是否全都不同。

1.

```
char firstUniqChar(string s) {
    vector<int> s_map(26, 0);
    for(auto& c:s){
        s_map[c-'a']++;
    }
    for(auto& c:s){
        if(s_map[c-'a'] == 1) {return c;}
    }
    return ' ';
}
```

2.

```
int firstUniqChar(string s) {
    vector<int> s_map(26, 0);
    for(auto& c:s) {s_map[c-'a']++;}
    for(int i = 0; i <s.size(); i++){
        if(s_map[s[i]-'a']==1) {return i;}
    }
    return -1;
}
```

3.

```
bool isUnique(string astr) {
    vector<int> s_map(26, 0);
    for(auto& c:astr){
        if(s_map[c-'a'] > 0) {return false;}
        s_map[c-'a']++;
    }
    return true;
}
```

## 字符串运算

415. 字符串相加：给定两个字符串形式的非负整数 num1 和 num2，计算它们的和。
43. 字符串相乘：给定两个以字符串形式表示的非负整数 num1 和 num2，返回 num1 和 num2 的乘积，它们的乘积也表示为字符串形式。
227. 基本计算器 II 和 面试题 16.26. 计算器：给定一个包含正整数、加(+)、减(-)、乘(\*)、除(/)的算数表达式(括号除外)，计算其结果。

**提升：**按关键词【计算器】搜索，查看其他计算器相关的困难题。

1.

```
string addStrings(string num1, string num2) {
    int i1 = num1.size()-1, i2 = num2.size()-1;
    string ans = ""; int plus = 0, sum = 0;
    while(i1 >= 0 && i2 >= 0){
        sum = num1[i1] - '0' + num2[i2] - '0' + plus;
        if(sum > 9) {plus = 1; sum = sum - 10;}
        else {plus = 0;}
        ans += (sum + '0');
        i1--; i2--;
    }
    while(i1 >= 0){
        sum = num1[i1] - '0' + plus;
        if(sum > 9) {plus = 1; sum = sum - 10;}
        else {plus = 0;}
        ans += (sum + '0'); i1--;
    }
}
```

2.

```
while(i2 >= 0){
    sum = num2[i2] - '0' + plus;
    if(sum > 9) {plus = 1; sum = sum - 10;}
    else {plus = 0;}
    ans += (sum + '0'); i2--;
}
if(plus) {ans += '1'};
if(ans.size() == 0) {return "0";}
reverse(ans.begin(), ans.end());
return ans;
```

2.

```

string multiply(string num1, string num2) {
    if(num1 == "0" || num2 == "0") {return "0";}
    string big, small;
    if(num1.size() > num2.size()) {big = num1; small = num2;}
    else {big = num2; small = num1;}
    string ans = "";
    int mul = 0, plus = 0;
    vector<string> mul_array;
    for(int i = small.size()-1; i>=0; i--){
        for(int j = big.size()-1; j>=0; j--){
            mul = (small[i]-'0')*(big[j]-'0') + plus;
            plus = mul/10; mul = mul%10;
            ans += (mul + '0');
        }
        if(plus>0) {ans += (plus + '0');}
        reverse(ans.begin(), ans.end());
        for(int cnt = small.size()-1-i; cnt>0; cnt--) {ans += '0';}
        mul_array.emplace_back(ans);
        ans = ""; plus = 0;
    }
    ans = "";
    for(auto& m:mul_array){
        ans = addStrings(ans, m); 趣函数
    }
    return ans;
}

```

## 3. 使用了辅助栈

基于此，我们可以用一个栈，保存这些（进行删除运算后的）整数的值。对于加减号后的数字，将其直接压入栈中；对于乘除号后的数字，可以直接与栈顶元素计算，并替换栈顶元素为计算后的结果。

具体来说，遍历字符串，并用变量 `preSign` 记录每个数字之前的运算符。对于第一个数字，其之前的运算符视为加号。每次遇到数字未尾时，根据 `preSign` 来决定计算方式：

- 加号：将数字压入栈；
- 减号：将数字的相反数压入栈；
- 乘除号：计算数字与栈顶元素，并将栈顶元素替换为计算结果。

```

int calculate(string s) {
    stack<int> st;
    char preSign = '+';
    int num = 0;
    for(int i = 0; i<s.size(); i++){
        if(isdigit(s[i])){
            num = num*10 + (s[i]-'0');
        }
        if(!isdigit(s[i]) && s[i] != ' ') || i == s.size()-1{
            switch(preSign){
                case '+':
                    st.push(num); break;
                case '-':
                    st.push(-num); break;
                case '*':
                    st.top() *= num; break;
                case '/':
                    st.top() /= num; break;
            }
            preSign = s[i];
            num = 0;
        }
    }
    int ans = 0;
    while(!st.empty()){
        ans += st.top();
        st.pop();
    }
    return ans;
}

```

## 字符串转换

- 剑指 Offer 46. 把数字翻译成字符串**: 给定一个数字，我们按照如下规则把它翻译为字符串：0 翻译成 “a”，1 翻译成 “b”，……，11 翻译成 “l”，……，25 翻译成 “z”。一个数字可能有多个翻译。请编程实现一个函数，用来计算一个数字有多少种不同的翻译方法。
- 剑指 Offer 67. 把字符串转换成整数**: 写一个函数 `StrToInt`，实现把字符串转换成整数这个功能。不能使用 `atoi` 或者其他类似的库函数。
- 12. 整数转罗马数字**: 给定一个整数，将其转为罗马数字。
- 13. 罗马数字转整数**: 给定一个罗马数字，将其转换成整数。
- 273. 整数转换英文表示** 和 **面试题 16.08. 整数的英语表示**: 将非负整数 `num` 转换为其对应的英文表示。PS：高频困难题，量力而行。
- 面试题 05.02. 二进制数转字符串**: 二进制数转字符串。给定一个介于0和1之间的实数（如 0.72），类型为double，打印它的二进制表达式。如果该数字无法精确地用32位以内的二进制表示，则打印“ERROR”。

## 1. 回溯

```

int translateNum(int num) {
    int ans = dfs(num);
    return ans;
}

int dfs(int num){//从num的最后一一位开始考虑
    if(num == 0) {return 1;}
    if((num%100) < 26 && (num%100) >=10) {return dfs(num/100) + dfs(num/10);}
    else {return dfs(num/10);}
}

```

## 2.

首先，该函数会根据需要丢弃无用的开头空格字符，直到寻找到第一个非空的字符为止。

当我们寻找到的第一个非空字符为正或者负号时，则将该符号与之后面尽可能多的连续数字组合起来，作为该整数的正负号；假如第一个非空字符是数字，则直接将其与之后连续的数字字符组合起来，形成整数。

该字符串除了有效的整数部分之后也可能存在多余的字符，这些字符可以被忽略，它们对于函数不应该造成影响。

注意：假如该字符串中的第一个非空字符不是一个有效整数字符、字符串为空或字符串仅包含空白字符时，则你的函数不需要进行转换。

在任何情况下，若函数不能进行有效的转换时，请返回 0。

说明：

假设我们的环境只能存储 32 位大小的有符号整数，那么其数值范围为  $[-2^{31}, 2^{31} - 1]$ 。如果数值超过这个范围，请返回 INT\_MAX ( $2^{31} - 1$ ) 或 INT\_MIN ( $-2^{31}$ )。

暴力

## 3.

```
string intToRoman(int num) {
    vector<int> num_map1 = {1,10,100,1000};
    vector<int> num_map5 = {5,50,500};
    vector<char> c_map1 = {'I','X','C','M'};
    vector<char> c_map5 = {'V','L','D'};
    string ans = "";
    for(int i = num_map1.size()-1; i>=0; i--){
        int div = num/num_map1[i];
        if(div == 0) {continue;}
        if(div == 4){
            ans += c_map1[i]; ans += c_map5[i];
        }
        else if(div == 9){
            ans += c_map1[i]; ans += c_map1[i+1];
        }
        else if(div == 5){
            ans += c_map5[i];
        }
        else{
            if(div > 5){
                ans += c_map5[i];
                div -= 5;
            }
            while(div > 0){
                ans += c_map1[i];
                div--;
            }
        }
        num = num % num_map1[i];
    }
    return ans;
}
```

## 4.

```
int strToInt(string str) {
    if(str.size() == 0) {return 0;}
    int num = 0;
    int i = 0;
    long compare = 0;
    bool sign = true;
    while(i < str.size()){
        if(str[i] != ' ') {
            if(str[i] == '+') {sign = true; break;}
            else if(str[i] == '-') {sign = false; break;}
            else if(isdigit(str[i])) {return 0;}
            else {num = num*10 + (str[i] - '0'); break;}
        }
        i++;
    }
    i++;
    if(i == str.size() && num == 0) {return 0;}
    for(; i<str.size(); i++){
        if(isdigit(str[i])){
            compare = (long)num*10 + (str[i] - '0');
            if(sign && compare >= (long)INT_MAX) {num = INT_MAX; break;}//2147483647
            else if(!sign && compare >= -(long)INT_MIN) {num = INT_MIN; break;}//-2147483648
            else {num = num*10 + (str[i] - '0');}
        }
        else {break;}
    }
    if(!sign && num>0) {num = -num;}
    return num;
}
```

## 4.

```
int romanToInt(string s) {
    unordered_map<char, int> s_map = {{'I',1},{'V',5},{'X',10},{'L',50},{'C',100},{'D',500},{'M',1000}};
    int ans = 0;
    for(int i = 0; i<s.size(); i++){
        if(i<s.size()-1 && s_map[s[i]] < s_map[s[i+1]]) {ans -= s_map[s[i]];}
        else {ans += s_map[s[i]];}
    }
    return ans;
}
```

## 5.

```
string numberToWords(int num) {
    if(num == 0) {return "zero";}
    unordered_map<int, string> unit = {{1, ""}, {1000, "Thousand"}, {1000000, "Million"}, {1000000000, "Billion"}};
    vector<string> map1 = {"Zero", "One", "Two", "Three", "Four", "Five", "Six", "Seven", "Eight", "Nine"};
    vector<string> map2 = {"Ten", "Eleven", "Twelve", "Thirteen", "Fourteen", "Fifteen", "Sixteen", "Seventeen", "Eighteen", "Nineteen"};
    vector<string> map3 = {"Twenty", "Thirty", "Forty", "Fifty", "Sixty", "Seventy", "Eighty", "Ninety"};
    string ans = "";
    for(auto& u:unit){
        int div = num/u.first;
        if(div == 0) {continue;}
        translate(ans, div, u.first, map1, map2, map3);
        ans += u.second; ans += " ";
        num = num%u.first;
    }
    while(ans.back() == ' ') {ans.pop_back();}
    return ans;
}

void translate(string& ans, int div, int ufirst, vector<string>& map1, vector<string>& map2, vector<string>& map3){
    int n = 100, tmp = 0, flag = false;
    while(n>0){
        tmp = div/n;
        if(flag){
            ans += map2[tmp]; ans += " ";
        }
        else if(tmp > 0){
            if(n == 100) {ans += map1[tmp]; ans += " Hundred ";}
            else if(n == 10){
                if(tmp > 1) {ans += map2[tmp-2]; ans += " ";}
                else if(tmp == 1) {flag = true;}
            }
            else{
                ans += map1[tmp]; ans += " ";
            }
        }
        div = div%n;
        n = n/10;
    }
}
```

6.

```

string printBin(double num) {
    string ans = "0.";
    for(int i = 0; i<32; i++){
        if(num == 0) {return ans;}
        num = num*2;
        if(num >= 1){
            ans += '1';
            num = num-1;
        } else{
            ans += '0';
        }
    }
    return "ERROR";
}

```

将小数0.6875转换成二进制数实例

0.6875	
*	2
1.3750	K <sub>i</sub> =1
*	2
0.7500	K <sub>i</sub> =0
*	2
1.5000	K <sub>i</sub> =1
*	2
1.0000	K <sub>i</sub> =1
1.0000	低位

所以 0.6875<sub>(10)</sub>=0.1011B

其实这个问题很简单，我们再拿0.6的二进制表示举例：1001 1001 1001 1001

文字描述：从左到右，v[i] \* 2^(i-1), i为从左到右的index, v[i]为该位的值，直接看例子，很直接的

$$0.6 = 1 * 2^{-1} + 0 * 2^{-2} + 0 * 2^{-3} + 1 * 2^{-4} + \dots$$

## 字符串游戏

1. 657. 机器人能否返回原点：在二维平面上，有一个机器人从原点(0, 0)开始。给出它的移动顺序，判断这个机器人在完成移动后是否在(0, 0)处结束。
2. 面试题 16.04. 井字游戏：设计一个算法，判断玩家是否赢了井字游戏。输入是一个N × N的数组棋盘，由字符" ", "X"和"O"组成，其中字符" "代表一个空位。
3. 面试题 16.22. 兰顿蚂蚁：编写程序来模拟蚂蚁执行的前K个动作，并返回最终的网格。
4. 6. Z字形变换：将一个给定字符串根据给定的行数，以从上往下、从左到右进行Z字形排列。

1.

```

bool judgeCircle(string moves) {
    int vertical = 0, horizon = 0;
    for(auto& c:moves){
        if(c == 'U') {vertical++;}
        else if(c == 'D') {vertical--;}
        else if(c == 'L') {horizon++;}
        else {horizon--;}
    }
    return (!vertical) && (!horizon);
}

```



```

string tictactoe(vector<string>& board) {
    size_t count_x = 0, count_o = 0, len = board.size();
    for (const auto& str : board) {
        for (const auto& ch : str) {
            if (ch == 'X') ++count_x;
            else if (ch == 'O') ++count_o;
        }
    }
    if (isWin('X', board, len)) return "X";
    if (isWin('O', board, len)) return "O";
    //是否没下满棋盘
    return count_x + count_o < len * len ? "Pending" : "Draw";
}

bool isWin(const char& ch, vector<string>& board, const size_t& len) {
    //横向
    for (size_t i = 0; i < len; ++i) {
        for (size_t j = 0; j < len; ++j) {
            if (board[i][j] != ch) break;
            else if (j == len - 1) return true;
        }
    }
    //纵向
    for (size_t j = 0; j < len; ++j) {
        for (size_t i = 0; i < len; ++i) {
            if (board[i][j] != ch) break;
            else if (i == len - 1) return true;
        }
    }
    //正对角线
    bool found = true;
    for (size_t i = 0; i < len; ++i) {
        if (board[i][i] != ch) {
            found = false;
            break;
        }
    }
    if (!found) return true;
    //逆对角线
    found = true;
    for (size_t i = 0; i < len; ++i) {
        if (board[i][len - 1 - i] != ch) {
            found = false;
            break;
        }
    }
    return found;
}

```

实际就是  
五子棋

3.

模拟即可，可以用一些小技巧：

1. 旋转方向：我们可以将每方向右下左的顺序(0, 1, 2, 3)保存下来。顺时针旋转时假设当前方向为右，则当前方向的指向的下一个方向为右-(p+1)。逆时针的下一个方向为p-(p+3)。

2. 转地颜色：我们用0表示地板为白色，1表示地板为黑色，则可以用异或1来进行这种操作：如果地板为白色 0^1 = 1 变成黑色  
如果地板为黑色 1^0 = 0 变成白色

```

vector<string> printSpiral(int K) {
    vector<vector<int>> dir = {{1,0}, {1,0}, {0,-1}, {-1,0}};/\right down left up
    vector<vector<int>> hash(3000, vector<int>(3000));//保存各地板颜色
    int p = 0;/当前位置指向的方向
    int x = 2000, y = 2000, minx = x, maxx = x, miny = y, maxy = y;
    vector<char> pos = {' ', 'O', ' ', 'U'};
    vector<char> board = {' ', ' ', ' ', 'X'};
    for(int i = 0; i<K; i++){//为逆时针
        int d = 1- i % 4;/为顺时针，d为逆时针
        if(hash[x][y]) (d = 3);
        hash[x][y] ~= 1;
        p = (p+d) % 4;
        x += dir[p][0]; y += dir[p][1];
        minx = min(minx, x); maxx = max(maxx, x);
        miny = min(miny, y); maxy = max(maxy, y);
    }
    vector<string> ans;
    for(int i = minx; i<=maxx; i++){
        string temp = "";
        for(int j = miny; j<=maxy; j++){
            if(i == x && j == y) {temp += pos[p];} //最后落脚的地方
            else {temp += board[hash[i][j]];}
        }
        ans.emplace_back(temp);
    }
    return ans;
}

```

```

4. string convert(string s, int numRows) {
    if(numRows == 1) {return s;}
    int numColumn = (s.size() / (numRows * 2 - 2) + 1) * (1 + numRows - 2);
    vector<vector<char>> s_map(numRows, vector<char>(numColumn, ' '));
    int flag = 0;
    int x = -1, y = 0;
    int cnt = 0;
    while(cnt < s.size()){
        if(flag%2){ // 奇数行
            x--; y++;
            if(x == 0) {flag++; x++; continue;}
            s_map[x][y] = s[cnt];
        } else{ // 偶数行
            x++;
            if(x == numRows) {flag++; x--; continue;}
            s_map[x][y] = s[cnt];
        }
        cnt++;
    }
    string ans = "";
    for(int i = 0; i < numRows; i++){
        for(int j = 0; j < numColumn; j++){
            if(s_map[i][j] != ' ') {ans += s_map[i][j];}
        }
    }
    return ans;
}

```

## 括号

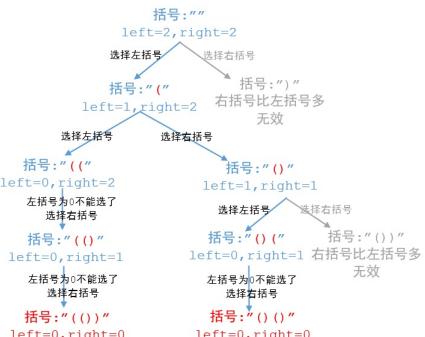
可以直接按关键词【括号】搜索，以下仅列举一些比较常见的。

- 面试题 08.09. 括号和22. 括号生成：数字n代表生成括号的对数，请你设计一个函数，用于能够生成所有可能的并且有效的括号组合。
20. 有效的括号：给定一个只包括'(', ')', '{', '}', '[', ']'的字符串，判断字符串是否有效。
856. 括号的分数：给定一个平衡括号字符串 S，按一定规则计算该字符串的分数。
32. 最长有效括号：给定一个只包含'(' 和 ')' 的字符串，找出最长的包含有效括号的子串的长度。PS：困难题量力而行。
1249. 移除无效的括号：给你一个由'('、')' 和小写字母组成的字符串 s。你需要从字符串中删除最少数目的'(' 或者 ')'（可以删除任意位置的括号），使得剩下的「括号字符串」有效。
301. 删除无效的括号：删除最小数量的无效括号，使得输入的字符串有效，返回所有可能的结果。



## 采用回溯

输入: n = 3  
输出: ["((()))","(()())","(())()","()((()))","()()()"]



```

vector<string> generateParenthesis(int n) {
    vector<string> ans;
    if(n <= 0) {return ans;}
    string path = "";
    dfs(n, n, path, ans);
    return ans;
}

void dfs(int left, int right, string path, vector<string>& ans){
    if(left == 0 && right == 0){ // 左右括号都不剩下了，说明找到了有效的括号
        ans.emplace_back(path);
        return;
    }
    //如果左括号只有剩余的时候才可以选，如果左括号的数量已经选完了，是不能再选左括号了。
    //如果选完了左括号我们是还可以选择右括号的。
    if(left < 0) {return;}
    if(right < left) {return;}
    dfs(left-1, right, path + '(', ans); //选择左括号
    dfs(left, right-1, path + ')', ans); //选择右括号
}

```

## 2. 采用哈希表及辅助栈

### 示例 2:

输入: s = "()[]{}"  
输出: true

### 示例 3:

输入: s = "]"  
输出: false

```
bool isValid(string s) {
    if(s.size() == 1) {return false;}
    unordered_map<char, char> hash = {('(', ')'), ('[', ']'), ('{', '}')};
    stack<char> st;
    for(auto& c:s){
        if(hash.count(c)){
            st.push(c);
        }
        else{
            if(st.empty()) {return false;}
            char tmp = st.top();
            st.pop();
            if(hash[tmp] != c) {return false;}
        }
    }
    if(!st.empty()) {return false;}
    return true;
}
```

## 3.

给定一个平衡括号字符串  $s$ ，按下述规则计算该字符串的分数：

- () 得 1 分。
- AB 得  $A + B$  分，其中 A 和 B 是平衡括号字符串。
- (A) 得  $2 * A$  分，其中 A 是平衡括号字符串。

- [0, 0] ()
- [0, 0, 0] ((
- [0, 1] ()()
- [0, 1, 0] ()()
- [0, 1, 0, 0] ()()()
- [0, 1, 1] ()()()
- [0, 3] ()()()
- [6] ()()()

```
int scoreOfParentheses(string s) {
```

```
stack<int> st;
st.push(0);
for(auto& c:s){
    if(c == '(') {st.push(0);}
    else{
        int num1 = st.top(); st.pop();
        int num2 = st.top(); st.pop();
        st.push(num2 + max(2*num1, 1));
    }
}
return st.top();
}
```

具体做法是我们始终保持栈底元素为当前已经遍历过的元素中「最后一个没有被匹配的右括号的下标」，这样的做法主要是考虑了边界条件的处理。栈里其他元素维护左括号的下标：

- 对于遇到的每个 '(', 我们将它的下标放入栈中
- 对于遇到的每个 ')', 我们先弹出栈顶元素表示匹配了当前右括号：
  - 如果栈为空，说明当前的右括号为没有被匹配的右括号，我们将其下标放入栈中来更新我们之前提到的「最后一个没有被匹配的右括号的下标」
  - 如果栈不为空，当前右括号的下标减去栈顶元素即为「以该右括号为结尾的最长有效括号的长度」

```
int longestValidParentheses(string s) {
```

```
int maxans = 0;
stack<int> stk;
stk.push(-1);
for (int i = 0; i < s.length(); i++){
    if (s[i] == '(') {stk.push(i);}
    else {
        stk.pop();
        if (stk.empty()) {stk.push(i);}
        else {maxans = max(maxans, i - stk.top());}
    }
}
return maxans;
}
```

## 5.

输入: s = "lee(t(c)o)de"  
输出: "lee(t(c)o)de"  
解释: "lee(t(co)de)"，"lee(t(c)ode)" 也是一个可行答案。

### 用cnt代替栈

```
string minRemoveToMakeValid(string s) {
```

```
int cnt = 0;
string ans = "";
for(int i = 0; i < s.size(); i++){
    if(s[i] == '(') {cnt++; ans += s[i];}
    else if(s[i] == ')'){
        if(cnt != 0) {ans += s[i]; cnt--;}
        else {ans += s[i];}
    }
}
while(cnt != 0){
    int pos = ans.find_last_of('(');
    ans.erase(pos, 1);
    cnt--;
}
return ans;
}
```



# 回溯

<https://leetcode.cn/problems/remove-invalid-parentheses/solution/gong-shui-san-xie-jiang-gua-hao-de-shi-f-a8u8/>

首先我们令左括号的得分为 1；右括号的得分为 -1。则会有如下性质：

1. 对于一个合法的方案而言，必然有最终得分为 0；
2. 搜索过程中不会出现得分值为 **负数** 的情况（当且仅当子串中某个前缀中「右括号的数量」大于「左括号的数量」时，会出现负数，此时不是合法方案）。

同时我们可以预处理出「爆搜」过程的最大得分：  $\max = \min(\text{左括号的数量}, \text{右括号的数量})$

枚举过程中出现字符串分三种情况：

- 左括号：如果增加当前 ( 后，仍为合法子串（即  $score + 1 \leq max$ ）时，我们可以选择添加该左括号，也能选择不添加；
- 右括号：如果增加当前 ) 后，仍为合法子串（即  $score - 1 \geq 0$ ）时，我们可以选择添加该右括号，也能选择不添加；
- 普通字符：直接添加。

使用 Set 进行方案去重，len 记录「爆搜」过程中的最大子串，然后只保留长度等于 len 的子串。

## 中大考研机试题

```

vector<string> removeInvalidParentheses(string s) {
    int n = s.size(), size = 0;
    set<string> s_set;
    int l = 0, r = 0;
    for(auto& c:s){
        if(c == '(') {l++;}
        else if(c == ')') {r++;}
    }
    int max = min(l, r); //字符串变有效后左右括号应有的对数
    dfs(s, 0, "", 0, max, s_set, n, size);
    vector<string> ans;
    for(auto& s:s_set){
        ans.emplace_back(s);
    }
    return ans;
}

void dfs(string& s, int i, string path, int score, int& max, set<string>& s_set, int& n, int& size){
    if(score < 0 || score > max) {return;}
    if(i == n){
        if(score == 0 && path.size() >= size){
            if(path.size() > size){
                s_set.clear();
                size = path.size();
            }
            s_set.insert(path);
        }
        return;
    }
    char c = s[i];
    if(c == '('){
        dfs(s, i+1, path + c, score + 1, max, s_set, n, size);
        dfs(s, i+1, path, score, max, s_set, n, size);
    }
    else if(c == ')'){
        dfs(s, i+1, path + c, score - 1, max, s_set, n, size);
        dfs(s, i+1, path, score, max, s_set, n, size);
    }
    else{
        dfs(s, i+1, path + c, score, max, s_set, n, size);
    }
}

```

## 回文

可以直接按关键词【回文】搜索，以下仅列举一些比较常见的。

1. **409. 最长回文串**：给定一个包含大写字母和小写字母的字符串，找到通过这些字母构造成的最长的回文串。
2. **125. 验证回文串**：给定一个字符串，验证它是否是回文串，只考虑字母和数字字符，可以忽略字母的大小写。
3. **680. 验证回文字符串 II**：给定一个非空字符串 s，最多删除一个字符。判断是否能成为回文字串。
4. **面试题 01.04. 回文排列**：给定一个字符串，编写一个函数判定其是否为某个回文串的排列之一。
5. **最长回文子串**：给定一个字符串 s，找到 s 中最长的回文子串。你可以假设 s 的最大长度为 1000。
6. **647. 回文子串**：给定一个字符串，你的任务是计算这个字符串中有多少个回文子串。具有不同开始位置或结束位置的子串，即使是由相同的字符组成，也会被视作不同的子串。
7. **516. 最长回文子序列**：给定一个字符串 s，找到其中最长的回文子序列，并返回该序列的长度。可以假设 s 的最大长度为 1000。
8. **214. 最短回文串**：给定一个字符串 s，你可以通过在字符串前面添加字符将其转换为回文串。找到并返回可以用这种方式转换的最短回文串。PS：困难题量力而行。

1.

```
int longestPalindrome(string s) {
    int ans = 0, div = 0;
    bool has_odd = 0;
    unordered_map<char, int> c_map;
    for(auto& c:s){
        c_map[c]++;
    }
    for(auto& m:c_map){
        div = m.second % 2;
        ans += (m.second - div);
        if(div) {has_odd = true;}
    }
    if(has_odd) {return ans + 1;}
    else {return ans;}
}
```

2.

```
bool isPalindrome(string s) {
    removeothers(s);
    if(s.size() <= 1) {return true;}
    for(int i = 0, j = s.size()-1; i<=j; i++, j--){
        if(s[i] != s[j]) {return false;}
    }
    return true;
}



原地移除非字母数字的字符，并把大写转为小写


void removeothers(string& s){
    int slow = 0, fast = 0;
    for(; fast < s.size(); fast++){
        if(isalpha(s[fast]) || isdigit(s[fast])){
            if(isalpha(s[fast]) && s[fast] < 'a'){
                s[fast] += 'a' - 'A';
            }
            s[slow] = s[fast];
            slow++;
        }
    }
    s.resize(slow);
}
```

3.

```
bool validPalindrome(string s) {
    return dfs(s, 0, s.size()-1, true);
}

bool dfs(string& s, int left, int right, bool valid){
    for(int i = left, j = right; i < j; i++, j--){
        if(s[i] != s[j]){
            if(valid) {return dfs(s, i+1, j, false) || dfs(s, i, j-1, false);}
            else {return false;}
        }
    }
    return true;
}
```

只有两层的回溯

4.

## 题目的逆解

```
bool canPermutePalindrome(string s) {
    unordered_map<char, int> cnt_map;
    for(auto& c:s){
        cnt_map[c]++;
    }
    bool has_odd = false;
    for(auto& n:cnt_map){
        if(n.second % 2 == 1){
            if(has_odd) {return false;}
            has_odd = true;
        }
    }
    return true;
}
```



## 找回文字串 $\Rightarrow$ 中心扩散法

首先往左寻找与当前位置相同的字符，直到遇到不相等为止。  
然后往右寻找与当前位置相同的字符，直到遇到不相等为止。  
最后双向扩散，直到左和右不相等。如下图所示：



<https://leetcode.cn/problems/longest-palindromic-substring/solution/zhong-xin-kuo-san-fa-he-dong-tai-gui-hua-by-reedfa/>

```
string longestPalindrome(string s) {
    if(s.size() == 0) {return "";}
    int s_size = s.size(), left = 0, right = 0, len = 1;
    int maxstart = 0, maxsize = 0;
    for(int i = 0; i < s_size; i++){
        left = i-1; right = i+1;
        len = 1;
        while(left >= 0 && s[left] == s[i]){//向左扩散
            len++; left--;
        }
        while(right < s_size && s[right] == s[i]){//向右扩散
            len++; right++;
        }
        while(left >= 0 && right < s_size && s[left] == s[right]){//同时向两边扩散
            len += 2; left--; right++;
        }
        if(len > maxsize) {maxsize = len; maxstart = left+1;}
    }
    return s.substr(maxstart, maxsize);
}
```



## 找回文字串 $\Rightarrow$ 中心扩散法

```
int countSubstrings(string s) {
    int s_size = s.size(), left = 0, right = 0;
    int cnt = 0;
    for(int i = 0; i < s_size; i++){
        left = i; right = i;//由i单个字符开始向左右扩散
        while(left >= 0 && right < s_size && s[left] == s[right]){
            cnt++; left--; right++;
        }
        left = i; right = i+1;//由i和i+1两个字符开始向左右扩散
        while(left >= 0 && right < s_size && s[left] == s[right]){
            cnt++; left--; right++;
        }
    }
    return cnt;
}
```

※要考虑到单个字符和双字符  
再指向左右扩散

## 7.

## 动态规划

```
int longestPalindromicSubseq(string s) {
    vector<vector<int>> dp(s.size(), vector<int>(s.size(), 0));
    for(int i = 0; i < s.size(); i++) {dp[i][i] = 1;}
    for(int i = s.size()-1; i >= 0; i--){
        for(int j = i+1; j < s.size(); j++){
            if(s[i] == s[j]) {dp[i][j] = dp[i+1][j-1] + 2;}
            else {dp[i][j] = max(dp[i+1][j], dp[i][j-1]);}
        }
    }
    return dp[0][s.size()-1];
}
```

<https://leetcode.cn/problems/longest-palindromic-subsequence/solution/dai-ma-sui-xiang-lu-dai-ni-xue-tou-dpzi-dv83q/>

## 8. 使用KMP求next数组

从暴力法可以看出，其实就是求 s 的「最长回文前缀」，然后在 rev\_s 的后缀中砍掉这个回文，再加到 s 前面。

这个最长前缀是回文的，它翻转之后等于它自己，出现在 rev\_s 的后缀，这不就是公共前后缀吗？KMP 的 next 数组记录的就是一个字符串的每个位置上，最长公共前后缀的长度。公共前后缀指的是前后缀相同。

因此，我们“制造”出公共前后缀，去套 KMP。

s: abab，则 s + '#' + rev\_s, 得到 str: abab#baba。

求出 next 数组，最后一项就是 str 的最长公共前后缀的长度，即 s 的最长回文前缀的长度。

s: abab    rev-s: baba     $\Rightarrow$  abab#baba  
            ↑                ↑                 ↑  
      最长回文前缀         最长公共前后缀

如果不加 #，'aaa' + 'aaa' 得到 'aaaaaa'，求出的最长公共前后缀是 6，但其实想要的是 3。

```
string shortestPalindrome(string s) {
    if(s.size() == 0) {return s;}
    string s_rev = s;
    reverse(s_rev.begin(), s_rev.end());
    string s_com = s + '#' + s_rev;
    vector<int> next(s_com.size(), 0);
    kmp(s_com, next);
    int maxlen = next[s_com.size()-1];
    s_rev = s_rev.substr(0, s_rev.size()-maxlen);
    return s_rev + s;
}
```