剑指offfer2

剑指 Offer 45. 把数组排成最小的数

思路

(排序) $O(nlogn)$

自定义排序规则，如果拼接 `ab` 可以比拼接 `ba` 更小的话，我们选择拼接 `ab`。

时间复杂度分析: 排序的时间复杂度为$O(nlogn)$。

c++代码

```
class Solution {
public:

    // 自定义排序规则，如果拼接ab 可以比拼接 ba更小的话，我们选择拼接ab
    static bool cmp(int a, int b)
    {
        string as = to_string(a), bs = to_string(b);
        return as + bs < bs + as;
    }
    string minNumber(vector<int>& nums) {
        sort(nums.begin(), nums.end(), cmp);
        string res;
        for(auto x : nums)
            res +=  to_string(x);
        return res;
    }
};
```

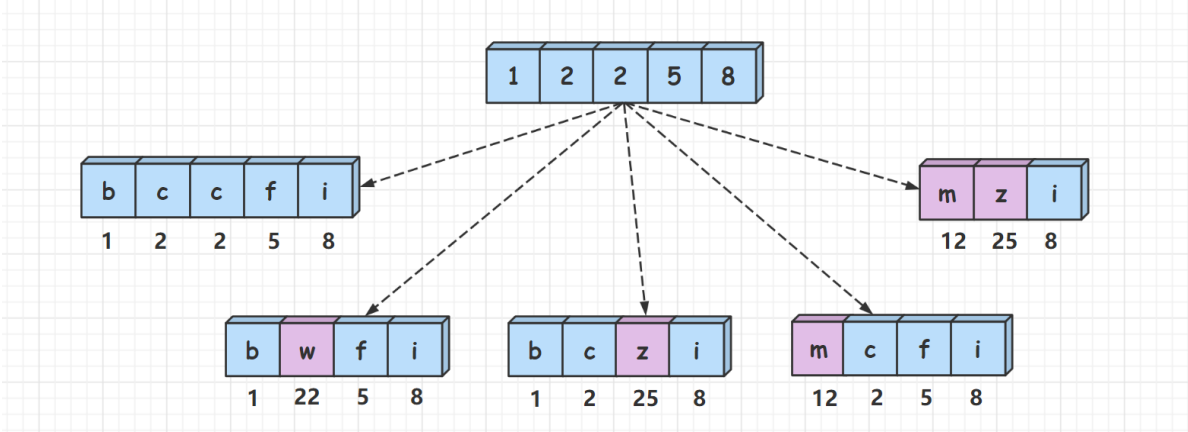
剑指 Offer 46. 把数字翻译成字符串

思路

(动态规划) $O(logn)$

给定我们一个数字 `num`，按照题目所给定的规则将其翻译成字符串，问一个数字有多少种不同的翻译方法。

样例:



我们先来理解一下题目的翻译规则，如样例所示，`num = 12258`，可以分为两种情况：

- 1、将每一位单独翻译，因此可以翻译成`"bccfi"`。
- 2、将相邻两位组合起来翻译（组合的数字范围在`10 ~ 25`之间），因此可以翻译成`"bwfi"`，`"bczi"`，`"mcfi"和"mzi"`。

两种情况是或的关系，互不影响，将其相加，那么`12258`共有`5`种不同的翻译方式。为了可以很方便的将数字的相邻两位组合起来，我们可以先将数字`num`转化成字符串数组`s[]`，下面来讲解动态规划的做法。

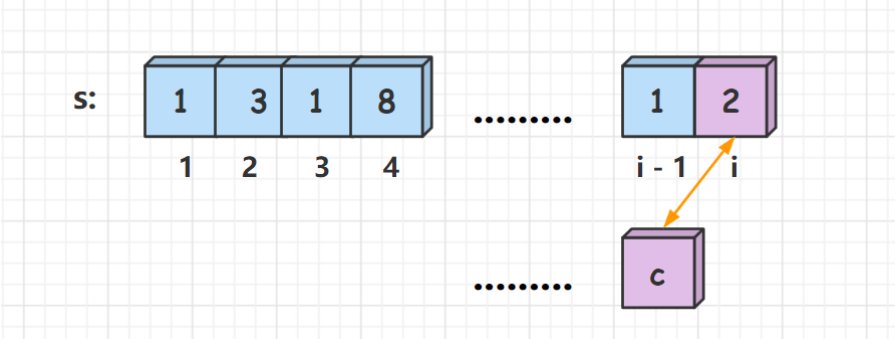
状态表示：

我们定义`f[i]`表示前`i`个数字一共有多少种不同的翻译方法。那么，`f[n]`就表示前`n`个数字一共有多少种不同的翻译方法，即为答案。

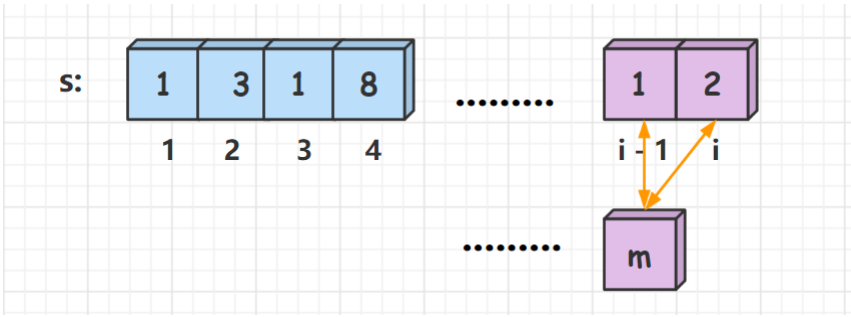
状态计算：

假设字符串数组为`s[]`，对于第`i`个数字，分成两种决策：

- 1、单独翻译`s[i]`。由于求的是方案数，如果确定了第`i`个数字的翻译方式，那么翻译前`i`个数字和翻译前`i - 1`个数的方法数就是相同的，即`f[i] = f[i - 1]`。(`s[]`数组下标从`1`开始)



- 2、将`s[i]`和`s[i - 1]`组合起来翻译(组合的数字范围在`10 ~ 25`之间)。如果确定了第`i`个数和第`i - 1`个数的翻译方式，那么翻译前`i`个数字和翻译前`i - 2`个数的翻译方法数就是相同的，即`f[i] = f[i - 2]`。(`s[]`数组下标从`1`开始)



最后将两种决策的方案数加起来，**因此，状态转移方程为：** `f[i] = f[i - 1] + f[i - 2]`。

初始化：

`f[0] = 1`，翻译前`0`个数的方法数为`1`。

为什么一个数字都没有的方案数是`1`？

`f[0]`代表翻译前`0`个数字的方法数，这样的状态定义其实是没有实际意义的，但是`f[0]`的值需要保证边界是对的，即`f[1]`和`f[2]`是对的。比如说，翻译前`1`个数只有一种方法，将其单独翻译，即`f[1] = f[1 - 1] = 1`。翻译前两个数，如果第`1`个数和第`2`个数可以组合起来翻译，那么`f[2] = f[1] + f[0] = 2`，否则只能单独翻译第`2`个数，即`f[2] = f[1] = 1`。因此，在任何情况下`f[0]`取`1`都可以保证`f[1]`和`f[2]`是正确的，所以`f[0]`应该取`1`。

实现细节：

我们将数字`num`转为字符串数组`s[]`，在推导状态转移方程时，假设的`s[]`数组下标是从`1`开始的，而实际中的`s[]`数组下标是从`0`开始的，为了一一对应，在取组合数字的值时，要把`s[i - 1]`和`s[i]`的值往前错一位，取`s[i - 1]`和`s[i - 2]`，即组合值`t = (s[i - 2] - '0') * 10 + s[i - 1] - '0'`。

在推导状态转移方程时，一般都是默认数组下标从`1`开始，这样的**状态表示**可以和实际数组相对应，理解起来会更清晰，但在实际计算中要错位一下，希望大家注意下。

时间复杂度分析： $O(logn)$，计算的次数是`nums`的位数，即`logn`，以`10`为底。

c++代码

```
class Solution {
public:
    int translateNum(int num) {
        string s = to_string(num);
        int n = s.size();
        vector<int> f(n + 1);
        f[0] = 1;  // 初始化
        for(int i = 1; i <= n; i++){
            f[i] = f[i - 1];
            if(i > 1){
                int t = (s[i - 2] - '0') * 10 + s[i - 1] - '0';
                if(t >= 10 && t <= 25)
                    f[i] += f[i - 2];
            }
        }
        return f[n];
    }
};
```

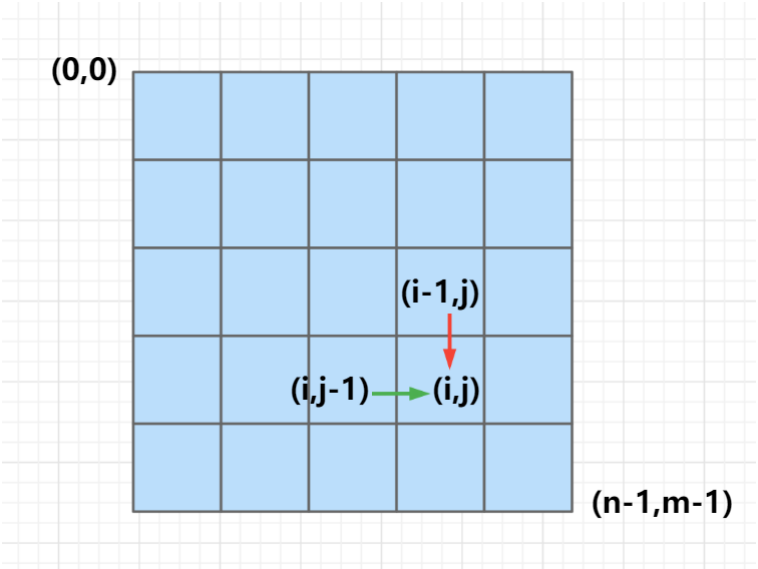
剑指 Offer 47. 礼物的最大价值

思路

(动态规划) $O(m*n)$

状态表示： `f[i,j]`表示从`(0,0)`走到`(i,j)`可以拿到的礼物最大价值。那么，`f[n - 1][m - 1]`就表示从棋盘左上角走到棋盘右下角可以拿到的礼物最大价值，即为答案。

状态转移：



由于限制了只会**向下走**或者**向右走**，因此到达`(i,j)`有两条路径

- 从上方转移过来，`f[i][j] = f[i-1][j] + grid[i][j]`
- 从左方转移过来，`f[i][j] = f[i][j-1] + grid[i][j]`

因此，**状态计算方程为：** `f[i][j] = max(f[i - 1][j], f[i][j - 1]) + grid[i][j]`，从向右和向下两条路径中选择礼物价值最大的转移过来，再加上`grid[i][j]`的值。

初始化： `f[0][0] = grid[0][0]`。

c++代码

```
class Solution {
public:
    int maxValue(vector<vector<int>>& grid) {
        int n = grid.size(), m = grid[0].size();
        vector<vector<int>>f(n + 1, vector<int>(m + 1));
        f[0][0] = grid[0][0];
        for(int i = 0; i < n; i++)
            for(int j = 0; j < m; j++){
                if(!i && !j) continue;
                if(i) f[i][j] = max(f[i][j], f[i - 1][j] + grid[i][j]);
                if(j) f[i][j] = max(f[i][j], f[i][j - 1] + grid[i][j]);
            }
        return f[n - 1][m - 1];
    }
};
```

剑指 Offer 48. 最长不含重复字符的子字符串、

思路

(双指针扫描) $O(n)$

定义两个指针 $i, j(i <= j)$，表示当前扫描到的子串是 $[i, j]$ (闭区间)。扫描过程中维护一个哈希表`unordered_map <chat,int>hash`，表示 $[i, j]$中每个字符出现的次数。

线性扫描时，每次循环的流程如下：

- 1.指针 j 向后移一位, 同时将哈希表中 `s[j]` 的计数加一:`hash[s[j]]++`;
- 2.假设 j 移动前的区间 $[i, j]$中没有重复字符，则 j 移动后，只有 `s[j]`可能出现`2`次。因此我们不断向后移动`i`，直至区间`[i,j]`中 `s[j]` 的个数等于`1`为止;

时间复杂度分析： 由于 `i`， `j` 均最多增加`n`次，且哈希表的插入和更新操作的复杂度都是 $O(1)$，因此，总时间复杂度 $O(n)$.

c++代码

```
class Solution {
public:
    int lengthOfLongestSubstring(string s) {
        unordered_map<char, int> hash; //存贮每个字符出现的次数
        int res = 0;
        for(int j = 0, i = 0; j < s.size(); j++){   //[j, i]
            hash[s[j]]++;
            while(hash[s[j]] > 1) hash[s[i++]]--;
            res = max(res, j - i + 1);
        }
        return res;
    }
};
```

剑指 Offer 49. 丑数

思路

(三路归并) $O(n)$

我们把只包含质因子 `2`、`3` 和 `5` 的数称作丑数，使用`vector<int> res` 存储每个丑数，且已知第一个丑数是 `1`，即 `res[0] = 1`。

用 `i`,`j`,`k`,三个指针分别指向三个序列：

- `i` 指向质因子包含 `2` 的所有数组成的序列 II。
- `j` 指向质因子包含 `3` 的所有数组成的序列 III。
- `k` 指向质因子包含 `5` 的所有数组成的序列 V。

初始状态下三个指针都是 `0`，指向第一个丑数 `res[0] = 1`。

三路归并，每次取 `res[i]*2`，`res[j]*3`，`res[k]*5`中的最小值，就是下一个丑数。
其中 `res[i]` 是序列 II 的第 `i`个数，那么 `res[i]*2` 就是第 `i + 1` 个数，`res[j]` 是序列 III 的第 `j` 个数，那么 `res[j]*3` 就是第 `j + 1` 个数，`res[k]` 是序列 V的第 `k` 个数，那么 `res[k]*5`就是第 `k + 1`个数。

`res[0] = 1` 不属于任何序列。

如果下一个丑数为 `res[i]*2`，则 `i`指针向往后移，如果 为`res[j]*3`，则 `j`指针往后移，如果为 `res[k]*5`，则 `k` 指针往后移。

细节：

如果下一个丑数即是 `2` 的倍数也是 `3` 的倍数，那么指针 `i` 和 `j` 都要往后移。

时间复杂度分析： 求第 `n` 个丑数，已知第一个丑数是 `1`，循环 `n - 1` 次即可求得，时间复杂度为 $O(n)$。

c++代码

```
class Solution {
public:
    int nthUglyNumber(int n) {
        vector<int> res;
        res.push_back(1);
        int i = 0, k = 0, j = 0;
        while(--n){
            int t = min(res[i] * 2, min(res[j] * 3, res[k] * 5));
```

```
            res.push_back(t);
            if(t % 2 == 0)  i++;
            if(t % 3 == 0)  j++;
            if(t % 5 == 0)  k++;
        }
        return res.back();
    }
};

```

剑指 Offer 50. 第一个只出现一次的字符

思路

(哈希) $O(n)$

1、定义一个`hash`表，存贮字符串`s`中每个字符出现的次数。

2、遍历整个字符串，如果字符串`s`中的某个字符出现的次数为`1`，则我们返回该字符。

3、如果没有，则返回空格。

时间复杂度分析： $O(n)$。

c++代码

```
class Solution {
public:
    char firstUniqChar(string s) {
        unordered_map<char, int> hash;
        for(char c : s) hash[c]++;
        for(char c : s){
            if(hash[c] == 1) return c;
        }
        return ' ';
    }
};
```

剑指 Offer 51. 数组中的逆序对

思路

(归并排序) $O(nlogn)$

归并排序模板：

```
const int maxn = 1e5 + 10;
int q[maxn], tmp[maxn];
void merge_sort(int q[], int l, int r)
{
    if (l >= r)  return;       //如果只有一个数字或没有数字，则无需排序
    int mid = (l + r ) / 2;
    merge_sort(q, l, mid);        //分解左序列
    merge_sort(q, mid + 1, r);    //分解右序列
    int k = l, i = l, j = mid + 1;
    while (i <= mid && j <= r)    //合并
    {
        if (q[i] <= q[j]) tmp[k++] = q[i++];
```

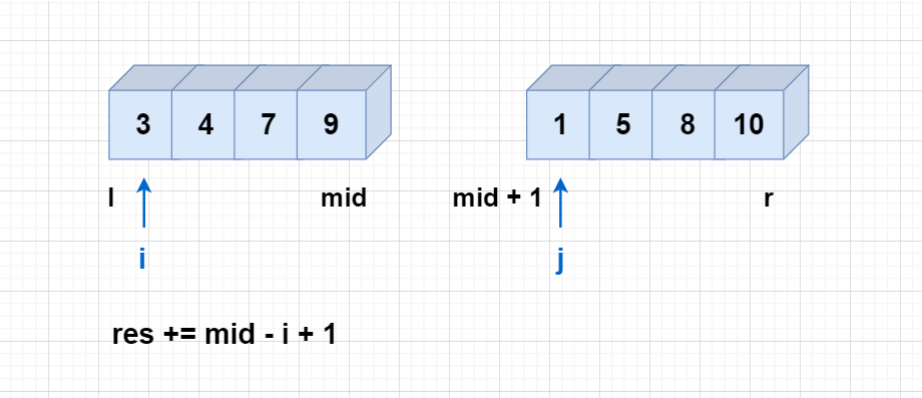
```
        else tmp[k++] = q[j++];
    }
    while (i <= mid) tmp[k++] = q[i++];     //复制左边子序列剩余
    while (j <= r)   tmp[k++] = q[j++];     //复制右边子序列剩余
    for (int i = l; i <= r; i++) q[i] = tmp[i];
}
```

在归并排序的合并操作中，我们假设左右两个区间元素为：

左边：`{3 4 7 9}` 右边：`{1 5 8 10}`

那么合并操作的第一步就是比较`3`和`1`，然后将`1`取出来放到辅助数组中，这个时候我们发现，右边的区间如果是当前比较的较小值，那么其会与左边剩余的数字产生逆序关系，也就是说`1`和`3`、`4`、`7`、`9`都产生了逆序关系，因此我们可以一下子统计出有`4`对逆序对。接下来`3`，`4`取下来放到辅助数组后，`5`与左边剩下的`7`、`9`产生了逆序关系，我们可以统计出`2`对。依此类推，`8`与`9`产生`1`对,那么总共有`4 + 2 + 1`对。这样统计的效率就会大大提高，便可较好地解决逆序对问题。

而在算法的实现中，我们只需略微修改原有归并排序，当右边序列的元素为较小值时，就统计其产生的逆序对数量，即可完成逆序对的统计。



时间复杂度分析： 归并排序的时间复杂度为$O(nlogn)$。

c++代码

```
class Solution {
public:
    int reversePairs(vector<int>& nums) {
        return merge(nums, 0, nums.size() - 1);
    }

    int merge(vector<int>&nums, int l, int r){
        if(l >= r) return 0; //序列中只有一个数
        int mid = l + r>> 1;
        int res = merge(nums, l, mid) + merge(nums, mid + 1, r);
        vector<int> tmp;
        int i = l, j = mid + 1;
        while(i <= mid && j <= r){
            if(nums[i] <= nums[j]) tmp.push_back(nums[i++]);
            else{
                res += mid - i + 1;
                tmp.push_back(nums[j++]);
            }
        }
        while(i <= mid) tmp.push_back(nums[i++]);
        while(j <= r)   tmp.push_back(nums[j++]);
        int k = l;
```

```
        for(int x : tmp) nums[k++] = x;
        return res;
    }
};
```

剑指 Offer 52. 两个链表的第一个公共节点

(链表，指针扫描) $O(n)$

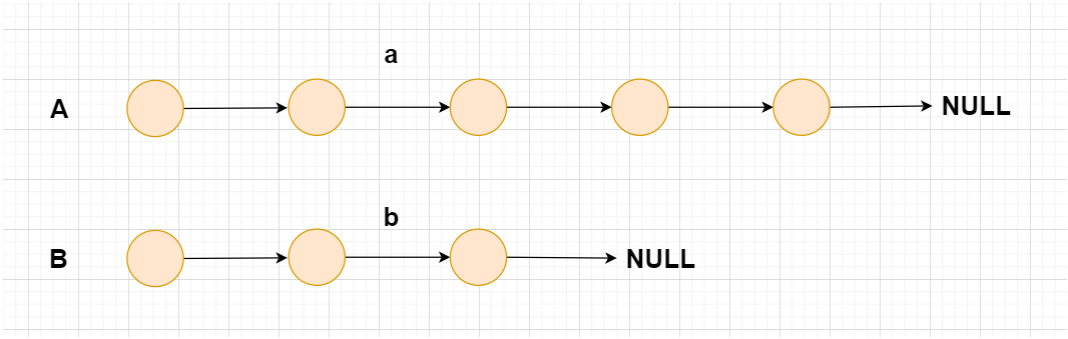
这题的思路很巧妙，我们先给出做法，再介绍原理。

算法步骤：

1. 用两个指针分别从两个链表头部开始扫描，每次分别走一步；
2. 如果指针走到`null`，则从另一个链表头部开始走；
3. 当两个指针相同时：
    - 如果指针不是`null`，则指针位置就是相遇点；
    - 如果指针是`null`，则两个链表不相交；

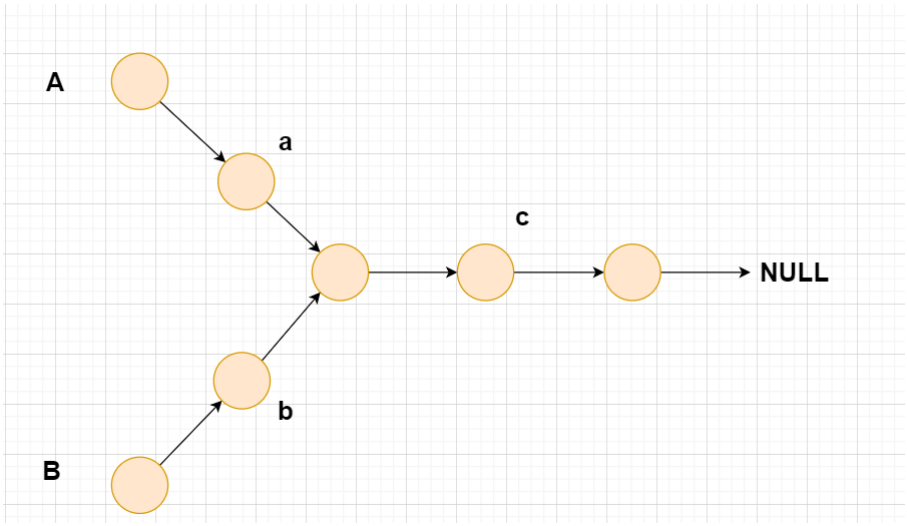
此题我们画图讲解，一目了然：

1、两个链表不相交：



`a`，`b` 分别代表两个链表的长度，则两个指针分别走 `a + b` 步后都变成 `null`。

2 、两个链表相交：



则两个指针分别走 `a + b + c` 步后在两链表交汇处相遇。

时间复杂度分析：每个指针走的长度不大于两个链表的总长度，所以时间复杂度是$O(n)$。

c++代码

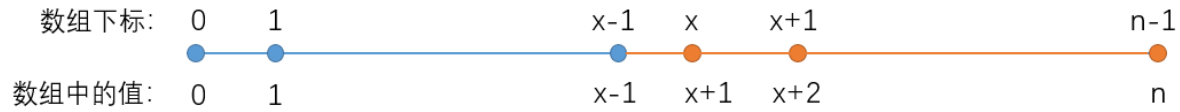
```cpp
/**
 * Definition for singly-linked list.
 * struct ListNode {
 *     int val;
 *     ListNode *next;
 *     ListNode(int x) : val(x), next(NULL) {}
 * };
 */
class Solution {
public:
    ListNode *getIntersectionNode(ListNode *headA, ListNode *headB) {
        auto pA = headA, pB = headB;
        while(pA != pB) {
            if(pA) pA = pA->next;
            else pA = headB;
            if(pB) pB = pB->next;
            else pB = headA;
        }
        return pA;
    }
};
```

剑指 Offer 53 - II. 0～n-1中缺失的数字

思路

(二分) $O(logn)$

假设数组中第一个缺失的数是 `x`，那么数组中的数如下所示:



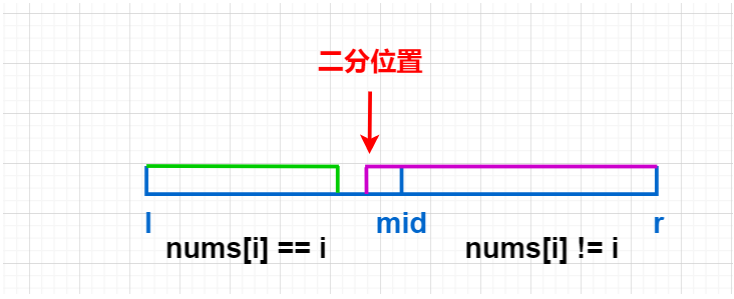
从中可以看出，数组左边蓝色部分都满足`nums[i] == i`，数组右边橙色部分都不满足`nums[i] == i`，因此我们可以二分出分界点 `x` 的值。

具体过程如下:

1、初始化`l = 0`， `r = nums.size() - 1`，二分`nums[i] !=i`的最左边界。

2、当`nums[mid] != mid`，说明答案在左半部分，往左边区域找，则`r = mid`。

3、当`nums[mid] == mid`，说明答案在右半部分，往右边区域找，则`l = mid + 1`。



4、当只剩下一个数时，就是缺失数字，我们返回`r`。

实现细节:

当所有数都满足`nums[i] == i`时，表示缺失的是`n`。

时间复杂度分析： 二分的时间复杂度是 $O(logn)$。

c++代码

```
class Solution {
public:
    int missingNumber(vector<int>& nums) {
        int l = 0, r = nums.size() - 1;
        while(l < r){
            int mid = (l + r) / 2;
            if(nums[mid] != mid) r = mid;
            else l = mid + 1;
        }
        if(nums[r] == r)  r++;  //缺失的是n
        return r;
    }
};
```

剑指 Offer 54. 二叉搜索树的第k大节点

思路

(dfs) $O(n)$

什么是二叉搜索树？

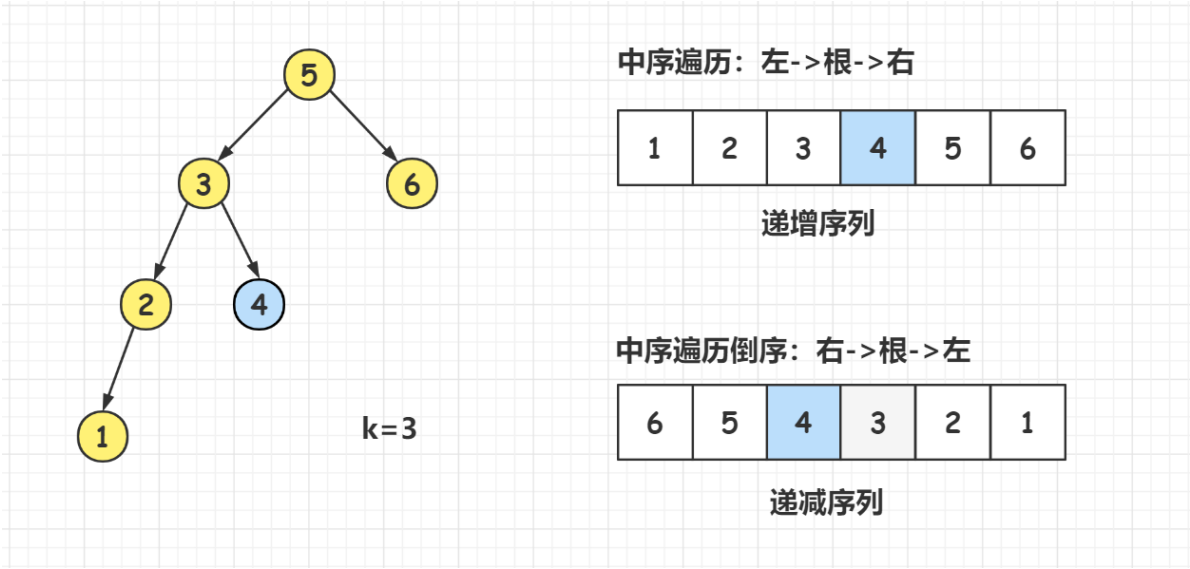
二叉搜索树是一棵有序的二叉树，所以我们也可以称它为二叉排序树。具有以下性质的二叉树我们称之为二叉搜索树：若它的左子树不为空，那么左子树上的所有值均小于它的根节点；若它的右子树不为空，那么右子树上所有值均大于它的根节点。它的左子树和右子树分别也为二叉搜索树。

二叉搜索树的中序遍历是：左=>根=>右； 二叉搜索树的中序遍历从小到大是有序的。

中序遍历模板

```
//打印中序遍历
void dfs(TreeNode* root )
{
    if(!root) return;
    dfs(root->left);    //左
    print(root->val);   //根
    dfs(root->right);   //右
}
```

如图所示



因此求二叉搜索树第 `k` 大的节点" 可转化为求 "二叉搜索树的中序遍历倒序的第 `k` 个节点"。

过程如下：

- 1、按照**右->根->左**的顺序（中序遍历倒序）遍历二叉树
- 2、我们每次遍历一个节点的时候就让 `k--`，当 `k` 减为 `0` 时，我们就找到了第 `k` 大的节点。

具体实现细节看代码。

时间复杂度分析： 每个节点最多只会被遍历 `1` 次，因此 `n` 个节点，时间复杂度为$O(n)$。

c++代码

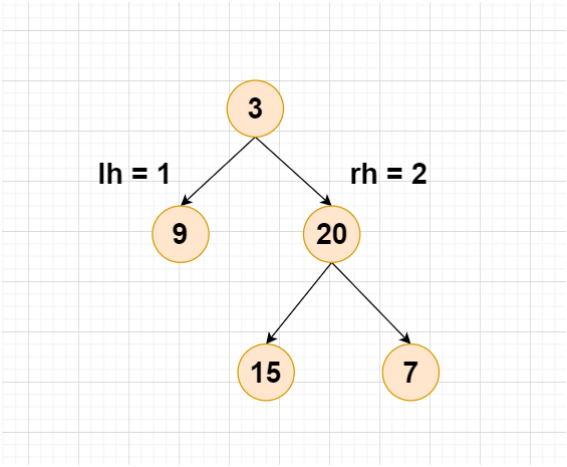
```cpp
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    int res;
    int kthLargest(TreeNode* root, int k) {
        dfs(root,k);
        return res;
    }
    void dfs(TreeNode* root ,int &k) //传引用 这里需要保证所有dfs函数共用一个k
    {
        if(!root) return;
        dfs(root->right,k);
        k--;
        if(!k) res = root->val;
        dfs(root->left,k);
    }
};
```

剑指 Offer 55 - I. 二叉树的深度

思路

(递归) $O(n)$

一棵二叉树的最大深度 `==` max(左子树最大深度，右子树最大深度) + `1`。



时间复杂度分析： $O(n)$

c++代码

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    //一棵二叉树的最大深度 == max(左子树最大深度, 右子树最大深度) + 1
    int maxDepth(TreeNode* root) {
        if(!root) return 0;
        int lh = maxDepth(root->left), rh = maxDepth(root->right);
        return max(lh, rh) + 1;
    }
};
```

剑指 Offer 55 - II. 平衡二叉树

思路

(递归) $O(n)$

二叉平衡二叉树的定义是：任意节点的左右子树的深度相差不超过`1`，而上一道题是求解二叉树的深度，因此我们可以在求解二叉树的深度过程中判断此树是不是平衡二叉树。

具体过程如下：

- 1、首先递归左右子树，求出左右子树的最大深度，我们记为`lh`，`rh`。
- 2、然后判断两棵子树是否是平衡的，即两棵子树的最大深度的差是否不大于`1`。

- 3、在递归的过程中记录每棵树的最大深度值，为左右子树的最大深度值加一，即`max(lh , rh) + 1`;
- 4、具体实现细节看代码

时间复杂度分析：每个节点仅被遍历一次，且判断的复杂度是 $O(1)$。所以总时间复杂度是 $O(n)$。

c++代码

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    bool res = true;
    bool isBalanced(TreeNode* root) {
        dfs(root);
        return res;
    }
    int dfs(TreeNode* root){   //求二叉树
        if(!root) return 0;
        int lh = dfs(root->left), rh = dfs(root->right);
        if(abs(lh - rh) > 1) res = false;  //判断
        return max(lh, rh) + 1;
    }
};
```

剑指 Offer 56 - I. 数组中数字出现的次数

思路

(位运算) $O(n)$

前置知识：

相同数异或为`0`，比如`a ^ a = 0`。

假设这两个只出现一次的数字分别为`x`和`y`，根据上述性质，我们将`nums`数组中的所有数都异或一遍，可以得到`x ^ y`。两个不同的数，其二进制表示中必然有一位是不同的，因此我们在`x ^ y`随便找到一个不同点，假设找到了为`1`的第`k`位。

我们可以根据第`k`位的不同，将序列分为两个集合，第`k`位为`1`的集合和第`k`位不是`1`的集合，其中`x`，`y`分别在这两个集合，且相同的元素是在同一个集合里面，于是将其转化成了求重复数字中的单个数值的问题。

我们将第`k`位为`1`的集合的所有数异或一遍，相同数相消，剩下的就是只出现一次的数，另一个数则为`x ^ y`再异或刚才我们求出的数。

时间复杂度分析： $O(n)$。

c++代码

```
class Solution {
```

```
public:
    // 相同数异或为0，a ^ a = 0
    /**
    异或得到 sum = x^y
    取 x与y中第k位为1的数
    将数分为两个集合，第k位为1的集合和第k位不是1的集合
    其中x y分别在这两个集合，且相同的元素是在同一个集合里面
    于是将其转化成了求重复数字中的单个数值的问题
    **/
    vector<int> singleNumbers(vector<int>& nums) {
        int sum = 0;
        for(int x : nums) sum ^= x; // sum = first ^ second;
        int k = 0;
        while(!(sum >> k & 1)) k++; //找到sum中为1的一位
        int first = 0;      //记录第一个数
        for(int x : nums){
            if(x >> k & 1)
                first ^= x;
        }
        return {first, sum ^ first};
    }
};

```

剑指 Offer 56 - II. 数组中数、字出现的次数 II

思路

(位运算) $O(n)$

如果一个数字出现 3 次，它的二进制每一位也出现的 3 次。如果把所有的出现 3 次的数字的二进制表示的每一位都分别加起来，那么每一位都能被 3 整除。 我们把数组中所有的数字的二进制表示的每一位都加起来。如果某一位能被3整除，那么这一位对只出现一次的那个数的这一肯定为 0。如果某一位不能被 3 整除，那么只出现一次的那个数字的该位置一定为 1。

因此，考虑二进制每一位上出现 0 和 1 的次数，如果出现 1 的次数为 3k + 1，则证明答案中这一位是 1。

具体过程：

- 1、定义 bit，从 0 枚举到 31，相当于考虑数字的每一位。
- 2、遍历数组 nums，统计所有数字 bit 位出现 1 的个数，记录到 cnt 中。
- 3、如果 bit 位 1 出现次数不是 3 的倍数，则说明答案在第 i 位是 1，否则说明答案的 bit 位是 0。

时间复杂度分析： 仅遍历 32 次数组，故时间复杂度为 $O(n)$。

c++代码

```
class Solution {
public:
    int singleNumber(vector<int>& nums) {
        int n = nums.size();
        int res = 0;
        for(int bit = 0; bit < 32; bit++){
            int cnt = 0; //统计所有数字bit位上1的个数
            for(int i = 0; i < nums.size(); i++){
                if(nums[i] >> bit & 1) cnt++;
```

```
            }
            if(cnt % 3 != 0) res += 1 << bit;
        }
        return res;
    }
};
```

剑指 Offer 57. 和为s的两个数字

思路

(双指针) $O(n)$

具体过程如下:

1、初始化两个指针 `i` , `j` 分别指向数组 `nums` 的左右两端。

2、循环搜索，当双指针相遇时跳出：

- 计算 `s = nums[i] + nums[j]` ;
- 如果 `s < target` ，则指针 `i` 向右移动;
- 如果 `s > target` ，则指针 `j` 向左移动;
- 如果 `s == target` ，则返回 `{nums[i],nums[j]}` ;

3、最后返回空数组，代表没有答案。

时间复杂度分析: $O(n)$。

c++代码

```
class Solution {
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        int i = 0, j = nums.size() - 1;
        while(i < j){
            if(nums[i] + nums[j] < target) i++;
            else if(nums[i] + nums[j] > target) j--;
            else return {nums[i],nums[j]};
        }
        return {};
    }
};
```

(哈希) $O(n)$

具体过程如下:

- 1、定义一个 `hash` 表，用来记录 `nums` 数组中每个数出现的次数
- 2、遍历整个 `nums` 数组，对于当前遍历的数字 `nums[i]` ，我们先在 `hash` 表中查找 `target - nums[i]` 是否存在。
    - 如果存在：直接返回 `{target - nums[i], nums[i]}` 。
    - 否则，将 `nums[i]` 加入 `hash` 表中 。

时间复杂度分析: $O(n)$。

c++代码

```
class Solution {
```

```
public:
    vector<int> twoSum(vector<int>& nums, int target) {
        vector<int> res;
        unordered_set<int> hash;
        for(int x : nums){
            if(hash.count(target - x)){
                res = vector<int>{target - x, x};
                return res;
            }
            hash.insert(x);
        }
        return res;
    }
};

```

剑指 Offer 57 - II. 和为s的连续正数序列

思路

(双指针) $O(n)$

我们定义两个指针`i`和`j`指针，将区间`[i,j]`看成滑动窗口，那么两个指针就分别表示滑动窗口的开始位置和结束位置，同时我们再维护一个`sum`变量用来存贮区间`[j,i]`连续序列的的和。通过不断调整两个指针来使得滑动窗口维护的区间和`sum`等于`target`，并记录答案。

过程如下：

- 1、我们定义两个指针`i`，`j`，初始化`i = 1`，`j = 1`，让两个指针都指向连续正数序列的开头，`i`指针用于扩展窗口，`j`指针用于收缩窗口。
- 2、枚举整个正数序列，当`sum < target`时，我们可以不断增加`j`使得滑动窗口向右扩展，同时`sum += j`，即`j++, sum += j`。
- 3、当`sum == target`并且滑动窗口的长度大于`1`时，将滑动窗口中的数记录到`res`中。
- 4、我们记录完一次合法的方案以后，就可以向右收缩滑动窗口，进行下一次合法方案的查找，同时`sum -= i`，即`sum -= i, i++`。

整个序列只需要枚举到`target/2`，即`target`的一半。

时间复杂度分析： $O(n)$。

c++代码

```
class Solution {
public:
    vector<vector<int>> findContinuousSequence(int target) {
        vector<vector<int>> res;
        int sum = 1;
        for(int i = 1, j = 1; i <= target / 2; i++){
            while(sum < target)  j++, sum += j;
            if(sum == target && j - i + 1 > 1){
                vector<int> path;
                for(int k = i; k <= j; k++) path.push_back(k);
                res.push_back(path);
            }
            sum -= i;
        }
        return res;

```

```
    }
};
```

剑指 Offer 58 - I. 翻转单词顺序

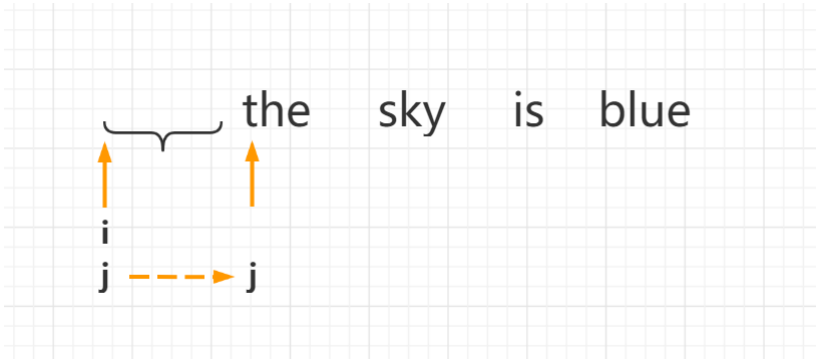
思路

对于样例 `"the sky is blue"` 分两步操作：

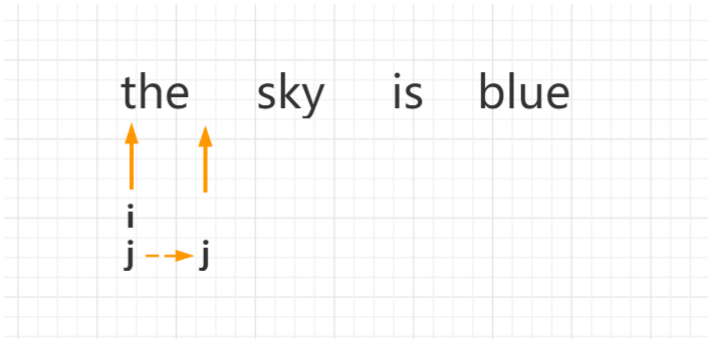
- 1、将字符串中的每个单词逆序，样例输入变为: `"eht yks si eulb"`；
- 2、将整个字符串逆序，样例输入变为：`"blue is sky the"`；

图示样例过程：

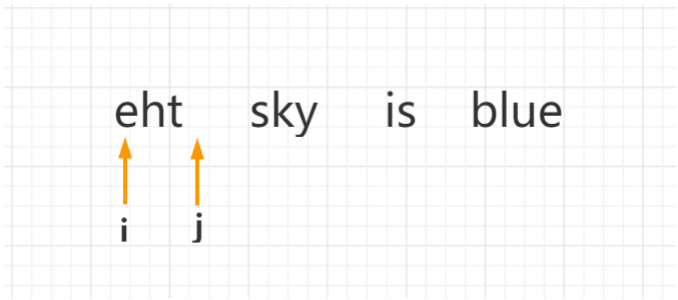
1、将 `i` 和 `j` 指针指向字符串的开头，并让 `j` 指针跳过字符串 `s` 单词的前导空格，指向单词的首非空字符。



2、再让 `i` 和 `j` 指针指向同一个位置，并让 `j` 指针跳过若干个非空字符指向单词后的第一个空格。



3、将 `i` 和 `j` 指针之间的单词翻转。



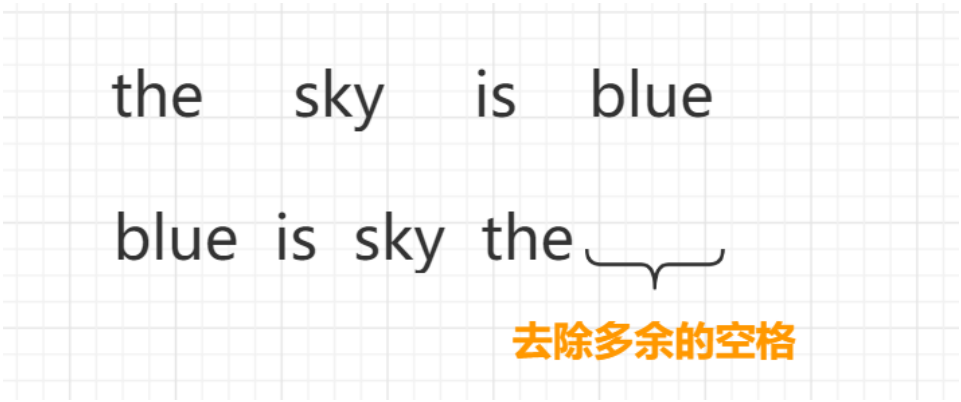
4、我们将翻转后的单词重新赋值给字符串 `s` 的前若干个字符（长度为单词的大小），重复上述过程。

5、最后将整个字符串翻转。

细节：

- 赋值之后要给每个单词后补个空格作为单词之间的分隔符。

- 最后要将字符串`s`之后多余的空格擦除。



- 具体实现细节看代码。

时间复杂度分析：整个字符串总共扫描两遍，所以时间复杂度是 $O(n)$。且每次翻转一个字符串时，可以用两个指针分别从两端往中间扫描，每次交换两个指针对应的字符，所以额外空间的复杂度是 $O(1)$。

c++代码

```
class Solution {
public:
    string reverseWords(string s) {
        int k = 0;
        for(int i = 0; i < s.size(); i++)
        {
            int j = i;
            while(j < s.size() && s[j] == ' ') j++; //j指针跳过单词的前导空格
            if(j == s.size()) break; //这里break是为了保证最后不会s[k++] = ' ', 避免行首多加一个空格
            i = j;
            while(j < s.size() && s[j] != ' ') j++;
            reverse(s.begin() + i, s.begin() + j);
            if(k)  s[k++] = ' '; //补个空格
            while( i < j) s[k++] = s[i++];
        }
        s.erase(s.begin() + k,s.end());//擦除多余的空格
        reverse(s.begin(),s.end());
        return s;
    }
};
```

剑指 Offer 58 - II. 左旋转字符串

思路

(字符串)

具体过程如下：

- 1、记录要转移的字符串`str`。
- 2、记录不转移的字符串`res`。
- 3、最后返回拼接到字符串`res + str`。

c++代码

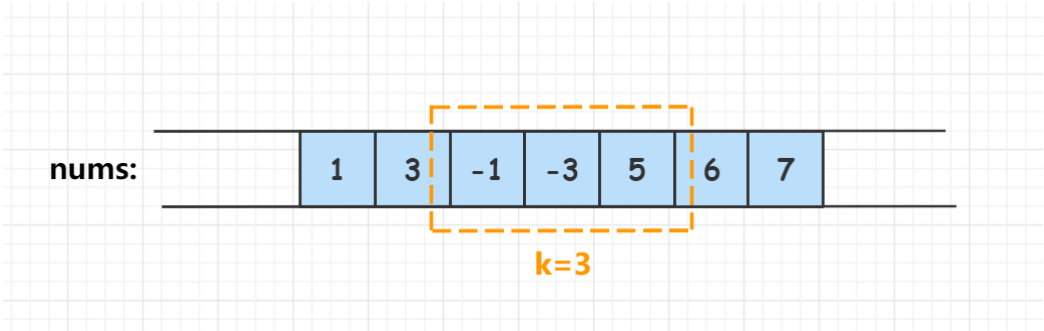
```cpp
class Solution {
public:
    string reverseLeftWords(string s, int n) {
        string str = s.substr(0, n); //记录要转移的字符串
        string res = s.substr(n);//记录不转移的字符串
        return res + str;
    }
};

```

剑指 Offer 59 - I. 滑动窗口的最大值

思路

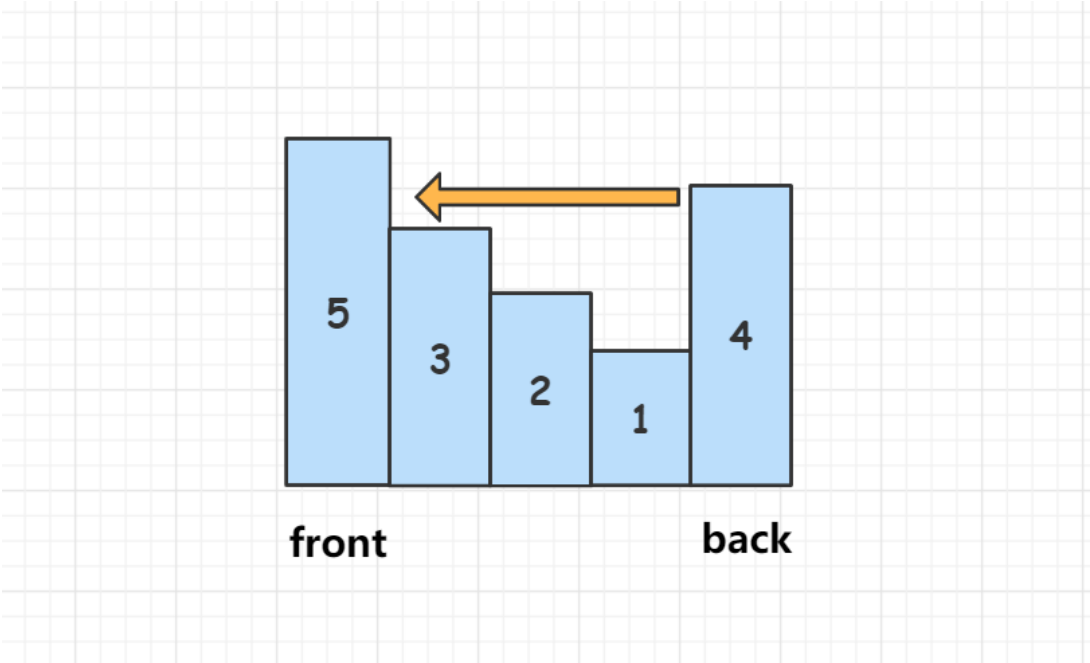
(单调队列) $O(n)$

首先，我们知道最直接的做法是模拟滑动窗口的过程，每向右滑动一次都遍历一遍窗口内的数字找最大的输出，这样的复杂度是$O(kn)$。考虑优化一下，窗口向右滑动的过程实际上就是将处于窗口的第一个数字删除，同时在窗口的末尾添加一个新的数字，这就可以用双向队列来模拟，每次把尾部的数字弹出，再把新的数字压入到头部，然后找队列中最大的元素即可。



如何快速找出滑动窗口中的最大值?

我们可以在队列中只保留那些可能成为窗口最大元素的数字，去掉那些不可能成为窗口中最大元素的数字。考虑这样一个情况，如果队列中进来一个较大的数字，那么队列中比这个数更小的数字就不可能再成为窗口中最大的元素了，而且这个大的数字是后进来的，一定会比之前早进入窗口的小的数字要晚离开窗口，那么那些早进入且比较小的数字就"永无出头之日"，所以就可以弹出队列。

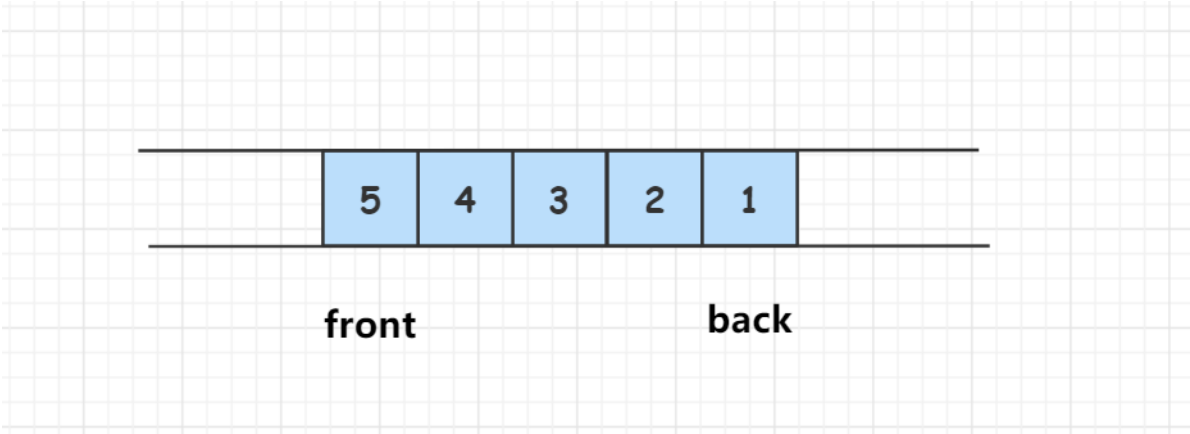


因此队列中的元素就会成保持一个单调递减的顺序，这样我们就维护了一个单调队列。

单调队列

单调队列是一个普通的双端队列，即队头和队尾都可以添加和弹出元素。单调队列顾名思义，队列中元素之间的关系具有单调性，此处的单调性分为单调递增与单调递减。

以单调递减队列为例：



(这里我们规定递减是指从队头到队尾是递减序列)

解题过程如下：

初始时单调队列为空，随着对数组的遍历过程中，每次插入元素前，需要考察两个事情：

- 1、合法性检查：队头下标如果距离 `i` 超过了 `k` ，则应该出队。
- 2、 单调性维护：如果 `nums[i]` 大于或等于队尾元素下标所对应的值，则当前队尾再也不可能充当某个滑动窗口的最大值了，故需要队尾出队。始终保持队中元素从队头到队尾单调递减。
- 3、如次遍历一遍数组，队头就是每个滑动窗口的最大值所在下标。

时间复杂度分析： 每个元素最多入队出队一次，复杂度为$O(n)$

c++代码

```cpp
class Solution {
public:
    vector<int> maxSlidingWindow(vector<int>& nums, int k) {
        deque<int>q; //双端队列
        vector<int>res;
        for(int i = 0; i < nums.size(); i++){
            while(q.size() &&  i - k + 1 > q.front())  q.pop_front(); //判断是否在滑动窗口范围内
            while(q.size() && nums[i] >= nums[q.back()]) q.pop_back();//维护单调递减队列
            q.push_back(i); //将当前元素插入队列
            if(i >= k - 1)  res.push_back(nums[q.front()]); //滑动窗口的元素达到了k个，才可以将其加入答案数组中
        }
        return res;
    }
};
```

剑指 Offer 59 - II. 队列的最大值

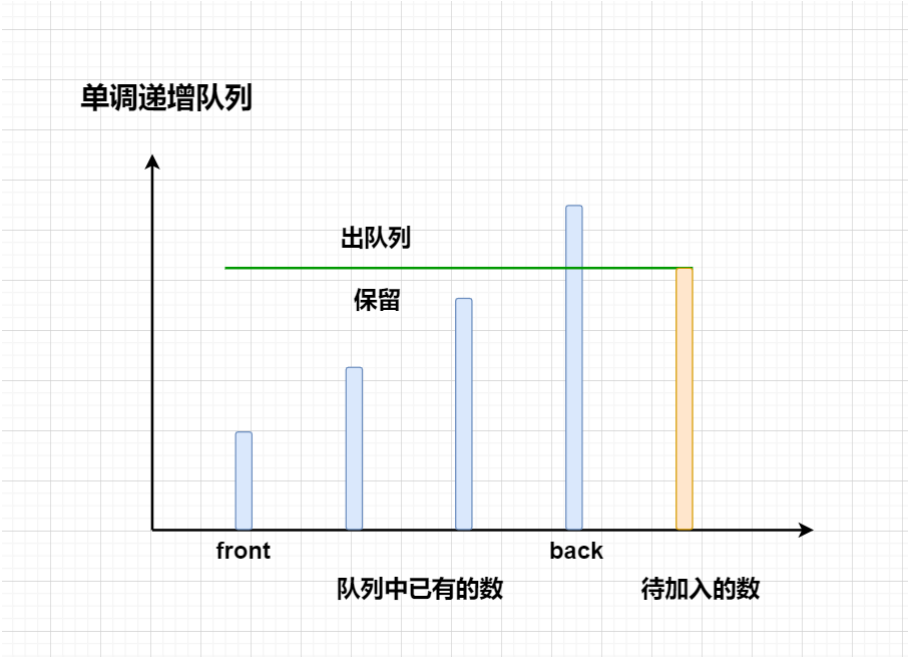
思路

(单调队列) $O(1)$

前置知识：

单调队列是一个普通的双端队列，即队头和队尾都可以添加和弹出元素。单调队列顾名思义，队列中元素之间的关系具有单调性，此处的单调性分为单调递增与单调递减。

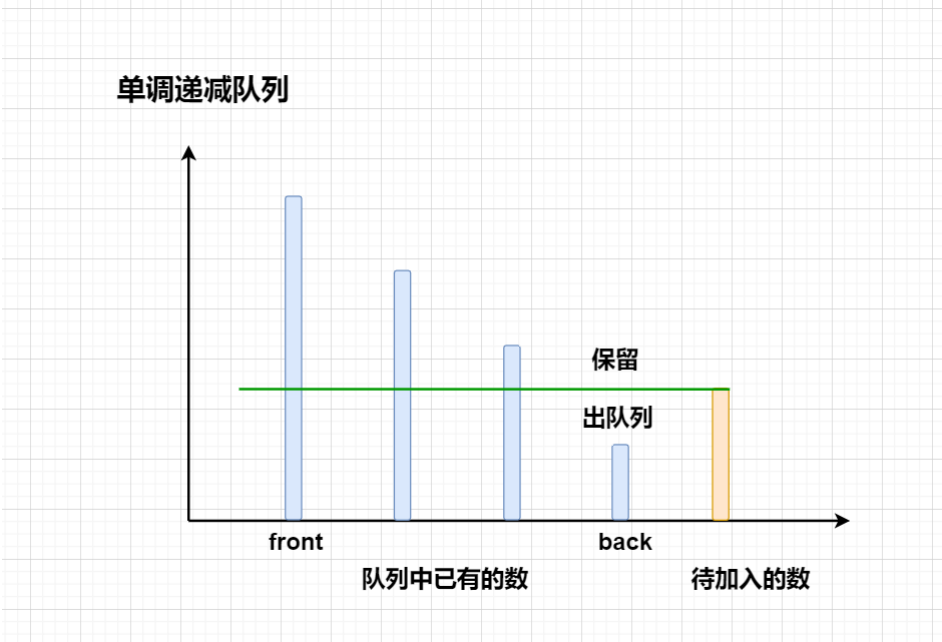
单调递增队列



对于队列内的元素来说：

1. 在队列内自己左边的数就是数组中左边第一个比自己小的元素。
2. 当被弹出时，遇到的就是数组中右边第一个比自己小的元素 。(只要元素还在队列中，就意味着暂时还没有数组中找到自己右侧比自己小的元素)
3. 队首元素为队列最小值。

单调递减队列



对于队列内的元素来说：

1. 在队列内自己左边的数就是数组中左边第一个比自己大的元素。
2. 当被弹出时，遇到的就是数组中右边第一个比自己大的元素 。(只要元素还在队列中，就意味着暂时还没有数组中找到自己右侧比自己大的元素)
3. 队首元素为队列最小值。

我们维护一个单调递减队列来保存队列中所有递减的元素 ， 随着入队和出队操作实时更新队列，这样队首元素始终就是队列中的最大值。

函数设计：

`max_value()`：

- 单调队列`deque`为空，则返回`-1`。
- 否则，返回`deque`队首元素。

`push_back()`：

- 将元素`value`加入`queue`。
- 如果队尾元素`back`小于`value`，我们就要不断弹出队尾元素，始终保持队中元素从队头到队尾单调递减，最后将元素`value`加入队队列`deque`。

`pop_back()`：

- 如果队列`queue`为空，则返回`-1`。
- 如果`deque` 首元素和 `queue` 首元素 **相等**，则将`deque`队首元素出队，以保持两队列元素一致。
- 返回队列`queue`队首元素。

时间复杂度分析： 方法的均摊时间复杂度均为 $O(1)$

c++代码

```
class MaxQueue {
public:

    queue<int> que;  //普通队列
    deque<int> deq;  //单调递减队列

    MaxQueue() {

    }

    int max_value() {
        return deq.empty() ? -1 : deq.front();
    }

    void push_back(int value) {
        que.push(value);
        while(deq.size() && deq.back() < value) deq.pop_back();
        deq.push_back(value);
    }

    int pop_front() {
        if(que.empty()) return -1;
        int val = que.front();
        if(val == deq.front()) deq.pop_front();
        que.pop();
        return val;
```

```
    }
};
```

剑指 Offer 60. n个骰子的点数

思路

(动态规划) $O(n^2)$

计算所有点数出现的概率，点数`x`出现的概率为： P(x) = x出现的次数 / 总次数

投掷 n个骰子，所有点数出现的总次数是 6^n。

掷1个骰子，点数之和范围为$1-6$。

掷2个骰子，点数之和范围为$2-6*2$。

因此，掷n个骰子，点数之和范围为$n-6*n$。

状态表示： `f[i][j]`表示投掷`i`个骰子，点数之和为`j`出现的次数。那么`f[n][x]`就表示投掷`n`个筛子，点数之和为`x`出现的次数。

状态计算： 我们依据最后一次投掷的点数划分集合，那么`f[i][j] += f[i - 1][j - k]`，`k`属于`{1, 2, 3, 4, 5, 6}`并且`j >= k`。

初始化： 初始化`f[1][1,2,3,4,5,6] = 1 = f[0][0]`。投掷`0`个骰子，点数之和为`0`只有一种方案。

时间复杂度分析： $O(n^2)$

c++代码

```
class Solution {
public:
    vector<double> dicesProbability(int n) {
        vector<vector<int>> f(n + 1, vector<int>(6 * n + 1, 0));
        f[0][0] = 1;
        for(int i = 1; i <= n; i++){
            for(int j = i; j <= 6 * i; j++)
                for(int k = 1; k <= 6; k++){
                    if(j >= k)
                        f[i][j] += f[i - 1][j - k]; //6次累加起来
                }
        }
        vector<double> res;
        int total = pow(6, n);
        for(int i = n; i <= 6 * n; i++){
            res.push_back(f[n][i] * 1.0/ total);
        }
        return res;
    }
};
```

剑指 Offer 61. 扑克牌中的顺子

思路

(数组，排序) $O(n)$

能组成顺子需要满足的两个条件是：

1. 除了`0`以外不能出现两个相同的数字。
2. 排序后两个相邻数字的差值不能大于`0`的个数。

具体过程如下：

- 1、将数组排序。
- 2、统计数组中`0`的个数。
- 3、遍历整个数组：
  - 如果在遍历过程中，出现了`0`以外的两个相同的数字或者相邻数字的差值大于`0`的个数，则返回`false`。
  - 维护`0`的个数，用总的`0`的个数减去用`0`填补空位的数量。
- 4、如果可以遍历到数组结尾，说明当前序列合法，返回`true`。

时间复杂度分析： $O(n)$

c++代码

```cpp
class Solution {
public:
    bool isStraight(vector<int>& nums) {
        int cnt = 0;
        sort(nums.begin(), nums.end());
        for(int x : nums){
            if(!x) cnt++; //统计0的个数
        }
        for(int i = 0; i < nums.size() - 1; i++){
            if(!nums[i]) continue;
            if(nums[i + 1] - nums[i] - 1> cnt) return false;
            if(nums[i + 1] == nums[i])  return false;
            cnt -= nums[i + 1] - nums[i] - 1;
        }
        return true;
    }
};
```

剑指 Offer 62. 圆圈中最后剩下的数字

思路

(递推，动态规划) $O(n)$

状态表示： `f[n][m]`表示`n`个人报数，每次报`m`的人被杀掉，最终胜利者的下标位置。

状态计算： 每杀掉一个人，则下一个人成为第一个报数的人，相当于把数组向前移动`m`位。若已知`n - 1`个人时，胜利者的下标位置为`f[n - 1][m]`，则`n`个人的时候，就是往后移动`m`，(因为有可能数组越界，超过的部分会被接到头上，所以还要模`n`)，即：`f[n][m] = (f[n - 1][m] + m) % n`，去掉报数的一维，则`f[i] = (f[i - 1] + m) % n;`

下图表示11个人从`1`开始报数，每次报`3`的被杀掉。

0	1	2	3	4	5	6	7
1	2	3	4	5	6	7	8
4	5	6	7	8	9	10	11
7	8	9	10	11	1	2	4
10	11	1	2	4	5	7	8
2	4	5	7	8	10	11	
7	8	10	11	2	4		
11	2	4	7	8			
7	8	11	2				
2	7	8					
2	7						
7							

时间复杂度分析： $O(n)$。

文章链接：https://blog.csdn.net/u011500062/article/details/72855826

c++代码

```
class Solution {
public:
    int lastRemaining(int n, int m){
        if (n == 1) return 0;
        vector<int> f(n + 1);
        f[1] = 0;
        for(int i = 2; i <= n ; i ++){ //枚举人数
            f[i] = (f[i - 1] + m) % i; //核心递推关系式
        }
        return f[n];
    }
};
```

剑指 Offer 63. 股票的最大利润

思路

(贪心) $O(n)$

由于只允许做一次股票买卖交易，因此我们用`minv`记录一只股票的最低价格，之后遍历整个`prices`数组，更新`minv`和`prices[i] - minv`的最大值。

时间复杂度分析： $O(n)$

c++代码

```cpp
class Solution {
public:
    int maxProfit(vector<int>& prices) {
        if(!prices.size()) return 0;
        int res = 0,minv = INT_MAX;
        for(int i = 0; i < prices.size(); i++){
            minv = min(minv, prices[i]);  //记录一只股票的最低价格
            res = max(res, prices[i] - minv);
        }
        return res;
    }
};

```

剑指 Offer 64. 求1+2+...+n

思路

(递归) $O(n)$

求`1 + 2 + ... + n`，但是不能使用乘除法、`for`、`while`、`if`、`else`、`switch`、`case`等关键字及条件判断语句，因此考虑递归

。

递归函数定义： `int sumNums(int n)`表示求`1 + 2 + ... + n`。

递归等价式： `sumNums(n) =  n + sumNums(n - 1)`。

递归边界： `n == 1`时，返回`1`。但是题目要求不能使用`if`，`while`等分支判断，可以考虑利用`&&`短路运算来终止判断。

时间复杂度分析： $O(n)$

c++代码

```cpp
class Solution {
public:
    int sumNums(int n) {
        int res = n;
        n >= 1 && (res += sumNums(n - 1));
        return res;
    }
};
```

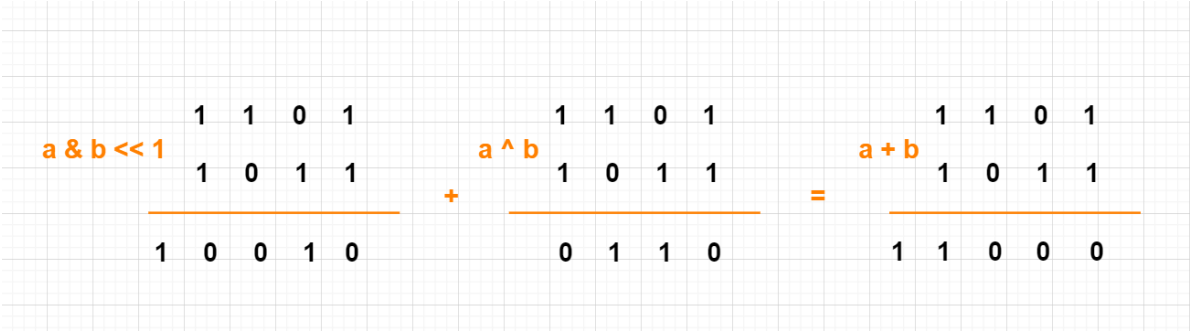
剑指 Offer 65. 不用加减乘除做加法

思路

(异或) $O(1)$

通过与运算与异或运算来实现加法运算：

- 1、计算两个数不算进位的结果 `(a ^ b)`
- 2、计算两个数进位的结果 `(a & b) << 1`
- 3、将两个结果相加，我们发现又要用到加法运算，那么其实我们重复上述步骤就行了，直到一个数变为`0`(不再进位)则运算全部完成。



时间复杂度分析： $O(1)$

c++代码

```
class Solution {
public:
    int add(int a, int b) {
        while(b){
            int x = a ^ b; //计算进位
            int y = (unsigned int)(a & b) << 1 ;  //计算进位，并防止溢出
            a = x;
            b = y;
        }
        return a;
    }
};
```

剑指 Offer 66. 构建乘积数组

思路

(数组) $O(n)$

由题意可知：$B[i] = A[0] \times A[1] \times \ldots \times A[i-1] \times A[i+1] \times \ldots \times A[n-1]$ 。

因此，我们可以通过两边遍历来实现：

- 1、第一遍正向遍历，求出 $B[i] = A[0] \times A[1] \times \ldots \times A[i-1]$。
- 2、第二遍反向遍历，求出$B[i]* = A[n-1] \times A[n-2] \times \ldots \times A[i+1]$。

最后我们返回 `B[]` 数组即可。

时间复杂度分析： 我们遍历了两次数组，因此时间复杂度为$O(n)$

c++代码

```
class Solution {
public:
    vector<int> constructArr(vector<int>& a) {
        int n = a.size();
        vector<int> res(n);
        for(int i = 0, p = 1; i < n; i++){  // 第一次算出 res[i] = a[0] *
a[1] * ... * a[i - 1]
            res[i] = p;
            p *= a[i];
        }
        for(int i = n - 1, p = 1; i >= 0; i-- ){
            res[i] *= p;                    // 第二次算出 res[i] *= a[n - 1] *
a[n - 2]*...* a[i + 1]
```

```
            p *= a[i];
        }
        return res;
    }
};
```

剑指 Offer 67. 把字符串转换成整数

思路

(模拟) $O(n)$

先来看看题目的要求：

- 1、**忽略所有行首空格**，找到第一个非空格字符，可以是 `‘+/-’` 表示是正数或者负数，紧随其后找到**最长的一串连续数字**，将其解析成一个整数。
- 2、整数后可能有**任意非数字字符**，请将其忽略。
- 3、如果整数大于 `INT_MAX`，请返回 `INT_MAX`；如果整数小于 `INT_MIN`，请返回 `INT_MIN`；

具体过程：

- 1、定义 `k = 0`，用 `k` 来找到第一个非空字符位置。
- 2、使用 `flag` 记录数字的正负性，`false` 表示正号，`true` 表示负号。
- 3、使用 `res` 来存贮结果，当 `str[k]` 为数字字符时进入 `while` 循环，执行 `res = res * 10 +str[k] - '0'`。
    - 根据 `flag` 判断，如果 `res` 大于 `INT_MAX`，则返回 `INT_MAX`；如果 `res * -1` 小于 `INT_MIN`，则返回 `INT_MIN`；
- 4、计算 `res`。

时间复杂度分析：字符串长度是 `n`，每个字符最多遍历一次，所以总时间复杂度是 $O(n)$。

c++代码

```
class Solution {
public:
    int strToInt(string str) {
        int k = 0;
        bool flag = false;

        while (k < str.size() && str[k] == ' ') k++;
        if (str[k] == '-') flag = true, k++;
        else if (str[k] == '+' ) k++;

        long long res = 0;
        while(k < str.size() && str[k] >= '0' && str[k] <= '9'){
            res = res * 10 + str[k] - '0';
            if (res > INT_MAX && !flag)     return  INT_MAX;
            if (res * -1 < INT_MIN && flag) return  INT_MIN;
            k++;
        }
        if(flag) res *= -1;
        return  res;
    }
};
```

剑指 Offer 68 - I. 二叉搜索树的最近公共祖先

思路

(递归) $O(n)$

二叉搜索树的定义：左子树`p->val`小于根节点`root->val`，根节点值小于右子树`q->val`。

因此可以利用二叉搜索树的特点，如果`p`、`q`的值都小于`root`，说明`p`，`q`肯定在`root`的左子树中；如果`p`，`q`都大于`root`，说明肯定在`root`的右子树中，如果一个在左一个在右，则说明此时的`root`记为对应的最近公共祖先。

递归过程中只有`3`种情况：

- `p->val <= root->val &&  q->val >=root->val`（结束，返回`root`值即结果）
- `p->val > root->val && q->val > root->val`（`root->right`递归）
- `p->val > root->val && q->val > root->val`（`root->left`递归）

时间复杂度分析： 每个节点最多只会被遍历一次，因此时间复杂度为$O(n)$。

c++代码

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q)
{
        if(!root) return NULL;
        if(p->val < root->val && q->val < root->val)  return
lowestCommonAncestor(root->left, p, q);
        if(p->val > root->val && q->val > root->val)  return
lowestCommonAncestor(root->right, p, q);
        return root;
    }
};
```

剑指 Offer 68 - II. 二叉树的最近公共祖先

思路

(递归) $O(n)$

`p`和`q`这两个节点共有三种情况：
1、`p`和`q`在`root`的子树中，且位于两侧。
2、`p = root`且`q`在`root`的左或右子树中。
3、`q = root`且`p`在`root`的左或右子树中。

考虑在左子树和右子树中查找这两个节点，如果两个节点分别位于左子树和右子树，则最低公共祖先为自己(root)，若左子树中两个节点都找不到，说明最低公共祖先一定在右子树中，反之亦然。考虑到二叉树的递归特性，因此可以通过递归来求得。

时间复杂度分析：需要遍历树，复杂度为 $O(n)$。

c++代码

```
/**
 * Definition for a binary tree node.
 * struct TreeNode {
 *     int val;
 *     TreeNode *left;
 *     TreeNode *right;
 *     TreeNode(int x) : val(x), left(NULL), right(NULL) {}
 * };
 */
class Solution {
public:
    TreeNode* lowestCommonAncestor(TreeNode* root, TreeNode* p, TreeNode* q) {
        if(!root) return NULL;  //没有找到，返回null
        if(root == p || root == q) return root; //找到其中之一，返回root
        TreeNode* left = lowestCommonAncestor(root->left, p, q);  //返回左子树查找节点
        TreeNode* right = lowestCommonAncestor(root->right,p ,q); //返回右子树查找节点
        if(left && right) return root;
        if(left) return left;
        else return right;
    }
};

```