# 算法

## 链表

### 反转单链表

* 从原链表的头部一个一个取节点并插入到新链表的头部，两个链表，一个新链表，一个原链表

|  |
| --- |
| ListNode\* ReverseList(ListNode\* pHead) {  ListNode\* newh = NULL;  for(ListNode\* p = pHead; p; )//p为工作指针  {  ListNode\* tmp = p -> next;//temp保存下一个结点  p -> next = newh;  newh = p;  p = tmp;  }  return newh;  } |

* 每次都将原第一个结点之后的那个结点放在新的表头后面。一个链表即可

## 字符串

### 旋转字符串

给定一个字符串，要求把字符串前面的若干个字符移动到字符串的尾部，如把字符串“abcdef”前面的2个字符'a'和'b'移动到字符串的尾部，使得原字符串变成字符串“cdefab”。请写一个函数完成此功能，要求对长度为n的字符串操作的时间复杂度为 O(n)，空间复杂度为 O(1)。

* 三步反转法 原串abcdef,k=3 => defabc

abc => cba，def => fed，cbafed => defabc

* 模拟字符移动，每次只将某个字符移动到它的正确位置，并用一个变量idx记录移动之后的位置，和cur记录被覆盖的值【即下一个要移动的值】

|  |  |  |
| --- | --- | --- |
| abcdef,idx=0,cur=’a’ | abcaef,idx=3,cur=’d’ | dbcaef,idx=0,cur=’a’ |

上面的转化就出现了问题，因为整个串长度为6，k=3,6恰好是3的倍数，这样就会到循环位置0 => 3 => 0 => 3，而不会遍历位置1 => 4 => 1 和 2

所以，我们需要另一个变量distance来让遍历的轨道发生变更,思路就是利用distance来记录一次循环将要发生，我们人为增加一次idx，让它到下一个轨道。

|  |  |  |
| --- | --- | --- |
| abcdef,idx=0,cur=’a’ | abcaef,idx=3,cur=’d’ | dbcaef,idx=1,cur=’b’ |

代码实现:

|  |
| --- |
| public class Solution {  public void rotate(int[] nums, int k) {  if(nums == null) return;  int len = nums.length;  /\*\*  \* idx:当前的位置  \* next:要替换的位置  \* cur:要替换的值  \* distance:发现循环发生的时刻，即只在一个轨道上循环  \*\*/  int idx = 0, cur = nums[0], next = 0, distance = 0;  int temp = 0;  //这个for循环只是为了保证替换发生了len次  for(int i=0; i<len; i++) {  next = (idx+k)%len;  temp = nums[next];  nums[next] = cur;  idx = next;  cur = temp;  if((distance=(distance+k)%len) == 0) {  idx = (idx+1)%len;  cur = nums[idx];  }  }  }  } |

类似题目：

|  |
| --- |
| 1、链表翻转。给出一个链表和一个数k，比如，链表为1→2→3→4→5→6，k=2，则翻转后  2→1→6→5→4→3，若k=3，翻转后3→2→1→6→5→4，若k=4，翻转后4→3→2→1→6→5，用程序实现。  2、编写程序，在原字符串中把字符串尾部的m个字符移动到字符串的头部，要求：长度为n的字符串操作时间复杂度为O(n)，空间复杂度为O(1)。  例如，原字符串为”Ilovebaofeng”，m=7，输出结果为：”baofengIlove”。  3、单词翻转。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，句子中单词以空格符隔开。为简单起见，标点符号和普通字母一样处理。例如，输入“I am a student.”，则输出“student. a am I”。 |

### 字符串包含

给定两个分别由字母组成的字符串A和字符串B，字符串B的长度比字符串A短。请问，如何最快地判断字符串B中所有字母是否都在字符串A里？

* HashTable
* 对字符串A，用位运算（26bit整数表示)计算出一个“签名”，再用B中的字符到A里面进行查找。

|  |
| --- |
| bool StringContain(string &a,string &b)  {  int hash = 0;  for (int i = 0; i < a.length(); ++i)  {  //精华，a[i]-‘A’得到字符的对应位置，1左移令字符对应位置为1，然后或操作  hash |= (1 << (a[i] - 'A'));  }  for (int i = 0; i < b.length(); ++i)  {  //做与操作，如果某一次与操作结果为0，就代表该字符未出现  if ((hash & (1 << (b[i] - 'A'))) == 0)  {  return false;  }  }  return true;  } |

### 字符串转换成整数

题意很明确，但注意事项很多，如下：

|  |
| --- |
| 1. 字符串为null，或空字符串时，返回整数0 2. 整数是有符号的，’+’ 或 ‘-’ 3. 去掉无效字符，从字符串第一个非空有效字符开始，到第一个无效字符(不包含)截止 4. 可能会出现整数溢出的情况，即大于Integer.MAX\_VALUE或小于Integer.MIN\_VALUE |

注意以上易错点之后，代码实现如下：

|  |
| --- |
| public class Solution {  public int myAtoi(String str) {  int index = 0, sign = 1, total = 0;  //1. Empty string  if(str.length() == 0) return 0;    //2. Remove Spaces  while(str.charAt(index) == ' ' && index < str.length())  index ++;    //3. Handle signs  if(str.charAt(index) == '+' || str.charAt(index) == '-'){  sign = str.charAt(index) == '+' ? 1 : -1;  index ++;  }    //4. Convert number and avoid overflow  while(index < str.length()){  int digit = str.charAt(index) - '0';  if(digit < 0 || digit > 9) break;    //check if total will be overflow after 10 times and add digit  if(Integer.MAX\_VALUE/10 < total || Integer.MAX\_VALUE/10 == total && Integer.MAX\_VALUE %10 < digit)  return sign == 1 ? Integer.MAX\_VALUE : Integer.MIN\_VALUE;    total = 10 \* total + digit;  index ++;  }  return total \* sign;  }  } |

上面的代码实现，思路清晰，效率高。

对比自己之前实现的代码：

|  |
| --- |
| public class Solution {  public int myAtoi(String str) {  if(str == null || str.equals("")) return 0;    String strTemp = str.trim();    int symbol = 1;  if(strTemp.charAt(0) == '-' || strTemp.charAt(0) == '+') {  symbol = strTemp.charAt(0) == '-' ? -1 : 1;  strTemp = strTemp.substring(1);  }    for(int i=0; i<strTemp.length(); i++) {  if(strTemp.charAt(i) < '0' || strTemp.charAt(i) > '9') strTemp = strTemp.substring(0, i);  }    double temp = 0.0;  boolean tag = false;  for(int i=0; i<strTemp.length(); i++) {  if(strTemp.charAt(i) != '0' || tag) {  temp = temp \* 10 + (strTemp.charAt(i) - '0');  tag = true;  }  }    if(temp\*symbol > Integer.MAX\_VALUE)  return Integer.MAX\_VALUE;  else if(temp\*symbol < Integer.MIN\_VALUE)  return Integer.MIN\_VALUE;    return (int)(temp\*symbol);  }  } |

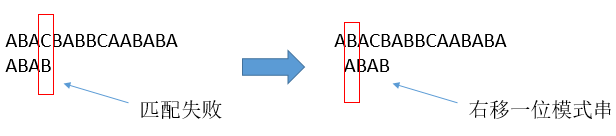
### KMP算法

给一个源字符串s : ABACBABBCAABABA 和一个模式pattern串p : ABABAC，判断源字符串S中是否出现模式串P。

问题很明确，也很容易想出简单的解题方法

* 暴力解法

只需将模式串沿着源字符串移动匹配，如果失败，就将模式串向后一个位置，再继续匹配。



* KMP算法

通过观察暴力解法可以观察到

两种思路，都是KMP算法

1. 利用前缀和后缀完全匹配最长的长度，如ABCAB，前后缀最长匹配长度为2。

这个思想主要是当在后面那个’B’后面的字符匹配失败时，我们就直接将模式串向后移动使前面AB跑到后面AB来，为什么呢，因为我们知道源字符串s里面肯定是有AB的，因为已经匹配到这里了，虽然在AB后面失败了，但是这个AB还是要匹配的，所以我们就要前后缀能匹配最长的长度，这样就可以直接将最长前缀移动到最长后缀那里

2）利用DFA有限状态机，模拟模式串匹配的过程，对于模式串的每个位置，源字符串出现任何字符都会对应一个状态，只有每个字符都完全匹配了，才会达到最终状态，才能匹配成功。

## 动态规划

### 1 KMP算法