

算法

```
#import "Algorithm.h"
#import <objc/runtime.h>

@implementation Algorithm

/
*****
*****
*****
```

其他算法

```
*****
*****
*****/
```

```
/**
 求和：11111 + 222222
 */
- (NSString *)add:(NSString *)str1 str2:
(NSString *)str2 {
    int i = str1.length - 1;
    int j = str2.length - 1;
    int temp = 0;
    NSMutableString *result = [NSMutableString
string];
    while (i >=0 || j >=0) {
        if (i >= 0) {
            temp += [str1
substringWithRange:NSMakeRange(i, 1)].intValue;
        }
        if (j >= 0) {
            temp += [str2
substringWithRange:NSMakeRange(i, 1)].intValue;
        }
    }
}
```

```

        result = [result
stringByAppendingString:[NSString
stringWithFormat:@"%d0", temp % 10]];
        temp /= 10;
        i--;
        j--;
    }
    return result;
}

/**
改变view上为button的颜色
*/
- (void)changeButtonColorIn:(UIView *)view {
    if (!view) {
        return ;
    }

    NSArray *sub = [view subviews];

    if ([sub count] == 0) {
        if ([view isKindOfClass:[UIButton
class]]) {
            UIButton *btn = (UIButton *)view;
            btn.backgroundColor = [UIColor
redColor];
        }
    }

    for (UIView *v in sub) {
        [self changeButtonColorIn:v];
    }
}

/**
子集合
*/
- (NSArray *)getSubsets:(NSArray *)nums {
    NSMutableArray *res = [NSMutableArray
array];
    // 空集

```

```

[res addObject:@[(0)]];

for (int i = 0; i < nums.count; i++) {
    // 取出一个元素
    NSInteger n = [nums[i] integerValue];
    // 子集的个数
    NSInteger size = res.count;
    NSInteger j = 0;
    // 遍历子集
    while (j < size) {
        // 取出每个子集
        NSMutableArray *sub =
[NSMutableArray arrayWithArray:res[j]];
        // 添加新元素
        [sub addObject:@(n)];
        // 再加入子集列表中
        [res addObject:sub];
        j += 1;
    }
}
return res;
}

```

/**

数组 排序最小

[12, 345, 2] -> [12,2,345]

* 解题思路：

* 先将整型数组转换成String数组，然后将String数组排序，最后将排好序的字符串数组拼接出来。关键就是制定排序规则。

* 排序规则如下：

* 若 $ab > ba$ 则 $a > b$ ，

* 若 $ab < ba$ 则 $a < b$ ，

* 若 $ab = ba$ 则 $a = b$ ；

* 解释说明：

* 比如 "3" < "31"但是 "331" > "313"，所以要将二者

拼接起来进行比较

比较两个字符串 s1, s2 大小的时候, 先将它们拼接起来, 比较 s1+s2, 和 s2+s1 那个大, 如果 s1+s2 大, 那说明 s2 应该放前面, 所以按这个规则, s2 就应该排在 s1 前面。

```
*/  
  
- (void)printMinNum:(NSMutableArray *)array {  
    NSMutableArray *nums = [array mutableCopy];  
    NSMutableString *str = [NSMutableString  
string];  
  
    for (int i = 0; i < nums.count; i++) {  
        for (int j = i + 1; j < nums.count; j++)  
{  
            NSInteger a = [[NSString  
stringWithFormat:@"%d%d", nums[i], nums[j]]  
integerValue];  
            NSInteger b = [[NSString  
stringWithFormat:@"%d%d", nums[j], nums[i]]  
integerValue];  
            if (a > b) {  
                [nums exchangeObjectAtIndex:i  
withObjectAtIndex:j];  
            }  
        }  
    }  
  
    for (int i = 0; i < nums.count; i++) {  
        [str appendString:[NSString  
stringWithFormat:@"%d", nums[i]]];  
    }  
  
    NSLog(@"%@", str);  
}  
  
/**  
    @[(1),(2),(3),(4)],  
    @[(5),(6),(7),(8)],
```

```

    @[@(9),@(10),@(11),@(12)]

    m * n
    */

- (void)logArray:(NSArray<NSArray *> *)array {
    NSInteger m = array[0].count; //4
    NSInteger n = array.count;    //3

    for (int i = 0; i < m; i++) {
        for (int k = 0; k < n; k++) {
            NSArray *temp = array[k];
            NSLog(@"==== %@", temp[i]);
            if (k == temp.count - 1 - i) {
                for (int j = i + 1; j <
temp.count; j++) {
                    NSLog(@"==== %@", temp[j]);
                }
            }
        }
    }
}

```

```

/
*****
*****
*****

```

排序算法

```

*****
*****
*****/

```

```

/**
冒泡排序 - 找最大，放最后

```

比较相邻的元素。如果第一个比第二个大，就交换他们两个。

对每一对相邻元素作同样的工作，从开始第一对到结尾的最后一对。这步做完后，最后的元素会是最大的数

针对所有的元素重复以上的步骤，除了最后一个

第一次比较后，找出最大的放到最后，第二次再从第一个开始找第二大的放到倒数第二。

```
*/  
- (NSArray *)bubbleSort:(NSArray *)items {  
    NSMutableArray *tempArray = [items  
mutableCopy];  
    NSInteger count = tempArray.count;  
    for (int i = 0; i < count - 1; i++) {  
        for (int j = 0; j < count - i - 1; j++)  
        {  
            if (tempArray[j + 1] < tempArray[j])  
            {  
                [tempArray  
exchangeObjectAtIndex:j withObjectAtIndex:j +  
1];  
            }  
        }  
    }  
    return tempArray;  
}
```

```
/**  
选择排序 - 找最小 放最前
```

从第一个元素开始，依次查找对比，找到最小的元素与第一个元素交换，

再从第二个元素开始找后面元素的最小值与第二个元素交换，以此类推，直到整个数组有序

找最小的，放到最前面，然后从第二个再开始找最小的

```
*/  
- (NSArray *)selectSort:(NSArray *)items {  
    NSMutableArray *tempArray = [items  
mutableCopy];  
    for (int i = 0; i < tempArray.count - 1; i+  
+) {  
        for (int j = i + 1; j < tempArray.count;
```

```

j++) {
    if (tempArray[j] < tempArray[i]) {
        [tempArray
exchangeObjectAtIndex:i withObjectAtIndex:j];
    }
}
return tempArray;
}

```

/*
插入排序 - 向前找

通过构建有序序列，对于未排序数据，在已排序序列中从后向前扫描，找到相应位置并插入。

将第一待排序序列第一个元素看做一个有序序列，把第二个元素到最后一个元素当成是未排序序列。

从头到尾依次扫描未排序序列，将扫描到的每个元素插入有序序列的适当位置

由数组的第2个位置开始比较，若果前方位置的元素比较大，则交换位置，若自己元素较大，而继续下一个元素，

如此排列，那么被操作的那个元素前方位置的所有元素皆为有序。

```

*/
- (NSArray *)insertSort:(NSArray *)items {
    NSMutableArray *tempArray = [items
mutableCopy];
    for (int i = 1; i < tempArray.count; i++) {
        // 取出当前值，和前一个的index
        int currentValue = [tempArray[i]
intValue];
        int preIndex = i - 1;
        // 当前值比前面的小，继续向前查找
        while (preIndex >= 0 && currentValue <
[tempArray[preIndex] intValue]) {
            // 把前一个数 放到当前位置
            tempArray[preIndex + 1] =
tempArray[preIndex];
            // 继续向前比较

```

```

        preIndex -= 1;
    }
    // 把当前数，放到前一个位置
    tempArray[preIndex + 1] =
    @(currentValue);
}
return tempArray;
}

```

/**
希尔排序

希尔排序是把记录按下标的一定增量分组，对每组使用直接插入排序算法排序；

随着增量逐渐减少，每组包含的关键词越来越多，当增量减至1时，整个文件恰被分成一组，算法便终止

```

*/
- (void)shellSort:(NSMutableArray *)items {
    int gap = (int)items.count / 2;
    while (gap >= 1) {
        for (int i = gap; i < items.count; i++)
        {
            int preIndex = i - gap;
            int currentValue = [items[gap]
intValue];
            while (preIndex >= 0 && currentValue
< [items[preIndex] intValue]) {
                items[preIndex + 1] =
items[preIndex];
                preIndex -= gap;
            }
            items[preIndex + 1] =
            @(currentValue);
        }
        gap = (int)items.count / 2;
    }
}

```

/**
快速排序 - 有基准

从数列中挑出一个元素，称为 "基准"， 重新排序数列，所有元素比基准值小的摆放在基准前面，所有元素比基准值大的摆在基准的后面（相同的数可以到任一边）。在这个分割之后，再以该基准为分界，分左右两边，分别挑选基准递归的进行排序

```
*/
- (void)quickSort:(NSMutableArray *)items left:
(int)left right:(int)right {
    // 数组不需要排序
    if (left >= right) return;

    // 第一个位置
    int l = left;
    // 最后一个位置
    int r = right;
    // 以第一个位置的元素作为基准值
    int pivot = [items[left] intValue];

    while (l != r) {
        // 从 右 -> 左 (从后向前走)，大于 base 的不
        // 动，小于的保存位置
        while (l < r && [items[r] intValue] >=
pivot)
            r--;

        // 从 左 -> 右 (从前向后走)，小于 base 的不
        // 动，大于的保存位置
        while (l < r && [items[l] intValue] <=
pivot)
            l++;

        // 交换位置
        if (l < r) {
            [items exchangeObjectAtIndex:l
withObjectAtIndex:r];
        }
    }

    // 把 base 归位
```

```

        [items exchangeObjectAtIndex:l
withObjectAtIndex:left];

        // 递归 左 到 分段点
        [self quickSort:items left:left right:l -
1];
        // 递归 分段点 到 右
        [self quickSort:items left:l + 1
right:right];
    }

/
*****
*****
*****

```

数组相关算法

```

*****
*****
*****/

```

/*
有序数组中找出和等于给定值的两个元素

从两头向中间开始查找

```

*/
- (NSArray *)towSum:(NSArray *)nums target:
(NSInteger)target {
    NSInteger i = 0;
    NSInteger j = nums.count - 1;

    while (i < j) {
        NSInteger sum = [nums[i] integerValue] +
[nums[j] integerValue];
        if (sum < target) {
            i++;
        } else if (sum > target) {
            j--;
        }
    }
}

```

```

        } else {
            return @[nums[i], nums[j]];
        }
    }
    return @[];
}

```

/**
 无序数组内部的两个元素的和为目标值

从第一个数开始，依次用差值进行查找
 */

```

- (NSArray *)twoSum2:(NSArray *)nums target:
(NSInteger)target {
    NSMutableArray *result = [NSMutableArray
array];
    for (int i = 0; i < nums.count; i++) {
        if ([nums containsObject:@(target -
[nums[i] intValue])]) {
            [result addObject:nums[i]];
            [result addObject:@(target -
[nums[i] intValue])];
        }
    }
    return result;
}

```

/**
 检测数组中是否包含重复的元素

*/

```

- (BOOL)isHasDuplicateNum:(NSArray *)nums {
    NSSet *set = [NSSet setWithArray:nums];
    return nums.count != set.count;
}

- (NSArray *)duplicateNum:(NSArray *)nums {
    NSMutableArray *result = [NSMutableArray
array];

    for (int i = 0; i < nums.count; i++) {
        NSArray *subArray = [nums

```

```

subarrayWithRange:NSMakeRange(i+1, nums.count -
i - 1)];

        if ([subArray containsObject:nums[i]]) {
            [result addObject:nums[i]];
        }
    }
    return result;
}

```

/**
出现次数超过数组长度一半的元素

放入字典，存取数目
*/

```

- (NSArray *)moreThanHalf:(NSArray *)items {
    NSMutableArray *result = [NSMutableArray
array];
    NSMutableDictionary *dict =
[NSMutableDictionary dictionary];

    for (int i = 0; i < items.count; i++) {
        if ([dict objectForKey:items[i]]) {
            NSInteger count = [[dict
objectForKey:items[i]] integerValue];
            [dict setObject:@(count + 1)
forKey:items[i]];

            if (count > items.count / 2) {
                [result addObject:items[i]];
            }

        } else {
            [dict setObject:@(1)
forKey:items[i]];
        }
    }

    return result;
}

```

/**
给定一个整数数组，除了某个元素外其余元素均出现两次。请找出
这个只出现一次的元素。

```
*/  
- (NSInteger)onceNumIn:(NSArray *)nums {  
    NSInteger result = 0;  
    for (int i = 0; i < nums.count; i++) {  
        result ^= [nums[i] integerValue];  
    }  
    NSLog(@"once %ld", result);  
    return result;  
}
```

/**
寻找连续数字数组中缺失的数字 [1,2,4]

连续数字串求和，然后减去每个元素 (首项+末项) * 项数 /
2

```
*/  
- (NSInteger)findMissing:(NSArray *)nums {  
    NSInteger n = nums.count;  
    NSInteger result = (n + 1) * n / 2;  
    for (int i = 0; i < nums.count; i++) {  
        result -= [nums[i] integerValue];  
    }  
    return result;  
}
```

/**
将所有的0移动到数组末尾

【1, 2, 0, 4, 0】 【1, 2, 4, 0, 0】

```
*/  
- (NSArray *)moveZero:(NSArray *)nums {  
    NSMutableArray *result = [NSMutableArray  
array];  
    for (int i = 0; i < nums.count; i++) {  
        if ([nums[i] integerValue] != 0) {  
            [result addObject:nums[i]];  
        }  
    }  
}
```

```

    }

    for (int i = 0; i < (nums.count -
result.count); i++) {
        [result addObject:@(0)];
    }

    return [result copy];
}

/**
两个数组的交点元素

以较少的为基准，判断，是否在较多的里面存在
*/

- (NSArray *)commonNums:(NSArray *)nums1 nums2:
(NSArray *)nums2 {
    if (nums1.count > nums2.count) {
        NSArray *temp = nums1;
        nums1 = nums2;
        nums2 = temp;
    }

    NSMutableArray *array = [NSMutableArray
array];
    for (int i = 0; i < nums1.count; i++) {
        if ([nums2 containsObject:nums1[i]]) {
            [array addObject:nums1[i]];
        }
    }
    return array;
}

/**
前K个最频繁的元素
[1,1,1,2,2,3] and k = 2, return [1,2].

```

桶排序，先用字典，统计出来每个数字的次数，然后以次数为下标，将元素放入数组中，再倒叙查找，找出次数最频繁的数字

```

    时间复杂度:  $O(n)$  空间复杂度: 很明显为  $O(n)$ 
    */
- (NSArray *)topMostIn:(NSArray *)nums k:
(NSInteger)k {
    // [1: 10];
    NSDictionary *dict = [NSDictionary
dictionary];
    for (int i = 0; i < nums.count; i++) {
        if ([dict objectForKey:nums[i]]) {
            [dict setValue:@(1 + [[dict
objectForKey:nums[i]] integerValue])
forKey:nums[i]];
        } else {
            [dict setValue:@(1) forKey:nums[i]];
        }
    }

    // 把次数作为下标，元素存放在桶中
    NSMutableArray *list = [NSMutableArray
array];
    [dict enumerateKeysAndObjectsUsingBlock:^(id
_Nonnull key, id _Nonnull obj, BOOL * _Nonnull
stop) {
        [list setObject:key atIndexSubscript:
(NSInteger)obj];
    }];

    NSMutableArray *result = [NSMutableArray
array];
    for(NSInteger i = list.count - 1; i >= 0 &&
result.count < k; i--){
        if(list[i] == nil) continue;
        [result addObject:list[i]];
    }
    return result;
}

/**
数组中第 k 大的元素
【1，2，3，3，2，6，8】

```

```

    */
- (NSInteger)kMaxIn:(NSArray *)nums k:
(NSInteger)k {
    // 去重
    NSMutableSet *set = [[NSMutableSet alloc]
initWithArray:nums];

    // 选择排序
    NSMutableArray *temp = [NSMutableArray
arrayWithArray:set.allObjects];
    for (int i = 0; i < temp.count - 1; i++) {
        for (int j = i + 1; j < temp.count; j++)
        {
            if ([temp[j] integerValue] <
[temp[i] integerValue]) {
                [temp exchangeObjectAtIndex:j
withObjectAtIndex:i];
            }
        }
    }

    // 拿出第K大的
    return [temp[k - 1] integerValue];
}

```

```

/**
合并两个有序的数组之后保持有顺序
*/
- (NSArray *)merge:(NSArray *)nums1 nums2:
(NSArray *)nums2 {
    NSMutableArray *mergeNums = [NSMutableArray
arrayWithCapacity:nums1.count + nums2.count];
    NSInteger i = 0;
    NSInteger j = 0;

    while (i < nums1.count && j < nums2.count) {
        if (nums1[i] < nums2[j]) {
            [mergeNums addObject:nums1[i]];
        } else {

```

```

        [mergeNums addObject:nums2[j]];
    }
    i++;
    j++;
}

if (i != nums1.count) {
    while (i < nums1.count) {
        [mergeNums addObject:nums1[i]];
        i++;
    }
}

if (j != nums2.count) {
    while (j < nums2.count) {
        [mergeNums addObject:nums1[j]];
        j++;
    }
}

return mergeNums;
}

```

/

```

*****
*****
*****

```

字符串算法

```

*****
*****
*****/

```

/**

交换局部字符集

@description 比如 将 "1234" 换成 "1324" 那么
startPosition = 1, endPosition = 2

```

@param pReverse 即将要个交换的字符串
@param startPosition 起始位置
@param endPosition 结束位置
@return 返回 这个返回值可有可无，因为 pReverse 是指针
*/
- (char *)reverseOptimizeFromSubString:(char
*)pReverse startPosition:
(NSUInteger)startPosition endPosition:
(NSUInteger)endPosition {
    // 只能是大于 0 的情况下 交换才有意义
    if ((endPosition - startPosition) < 1) {
        return pReverse;
    }
    // 开始交换
    for(NSUInteger j=startPosition;
j<(startPosition + ((endPosition+1)-
startPosition)/2); j++) {
        char cc = pReverse[j];
        NSUInteger k = endPosition - (j -
startPosition);
        pReverse[j] = pReverse[k];
        pReverse[k] = cc;
    }
    return pReverse;
}

- (NSString *)reverseStr:(NSString *)input {
    if (input.length < 1) {
        return input;
    }

    char *str = (char*)[input UTF8String];
    NSInteger end = input.length - 1;
    for (int i = 0; i < (end + 1)/2; i++) {
        char temp = str[i];
        str[i] = str[end - i];
        str[end - i] = temp;
    }
}

```

```

        return [NSString stringWithUTF8String:str];
    }

    /**
     字符串逆序
     */
    // 1234 -> 4321
    - (NSString *)invertStr:(NSString *)input {
        NSMutableString *temp = [NSMutableString
string];
        for (int i = 0; i < input.length; i++) {
            [temp appendString:[input
substringWithRange:NSMakeRange(input.length - 1
- i, 1)]];
        }
        return temp;
    }

    // hello word -> word hello
    - (NSString *)invertWordBySpace:(NSString
*)input {
        NSArray *words = [input
componentsSeparatedByString:@" "];
        NSMutableArray *newStrArray =
[NSMutableArray array];
        for (int i = 0; i < words.count; i++) {
            [newStrArray addObject:words[words.count
- 1 - i]];
        }
        NSString *newStr = [newStrArray
componentsJoinedByString:@" "];
        return newStr;
    }

    // hello word -> drow olleh
    - (NSString *)inverCirlce:(NSString *)input {
        NSArray *wordsArray = [input
componentsSeparatedByString:@" "];
        NSMutableArray *invertWordsArray =
[NSMutableArray array];
        for (int i = 0; i < wordsArray.count; i++) {

```

```

        NSString *word =
wordsArray[wordsArray.count - 1 - i];
        NSMutableString *invertWord =
[NSMutableString string];
        for (int j = 0; j < word.length; j++) {
            [invertWord appendString:[word
substringWithRange:NSMakeRange(word.length - 1 -
j, 1)]];
        }
        [invertWordsArray addObject:invertWord];
    }
    NSString *invertStr = [invertWordsArray
componentsJoinedByString:@" "];
    return invertStr;
}

```

/

```

*****
*****
*****

```

查找算法

```

*****
*****
*****/

```

/**

开平方

若N大于1,则从[1, N]开始, low = 1, high = N, mid = (low + high) / 2 开始进行数值逼近

若N小于1,则从[N, 1]开始, low = 0, high = N, mid = (low + high) / 2 开始进行数值逼近

*/

```

// - (CGFloat)sqrt:(CGFloat)n {
//     CGFloat low;

```

```

//      CGFloat height;
//      CGFloat middle;
//
//      if (n > 1) {
//          low = 1;
//          height = n;
//      } else {
//          low = n;
//          height = 1;
//      }
//
//      while (<#condition#>) {
//          <#statements#>
//      }
//}

```

```

/**
二分查找

```

```

数组需是有序的
*/
- (NSInteger)binarySearch:(NSArray *)nums
target:(NSInteger)target {
    NSInteger low = 0;
    NSInteger high = nums.count - 1;
    NSInteger middle = 0;

    while (low <= high && low <= nums.count - 1
&& high <= nums.count - 1) {
        middle = (low + high) / 2;
        if (target == [nums[middle]
integerValue]) {
            return middle;
        } else if (target < [nums[middle]
integerValue]) {
            high = middle - 1;
        } else {
            low = middle + 1;
        }
    }
}

```

```

        return -1;
    }

- (NSInteger)binarySearch:(NSArray *)nums left:
(NSInteger)low right:(NSInteger)high target:
(NSInteger)target {
    if (low > high || low > nums.count - 1 ||
high > nums.count - 1) {
        return -1;
    }
    NSInteger middle = (low + high) / 2;
    if ([nums[middle] integerValue] == target) {
        return middle;
    } else if (target < [nums[middle]
integerValue]) {
        return [self binarySearch:nums left:low
right:middle - 1 target:target];
    } else {
        return [self binarySearch:nums
left:middle + 1 right:high target:target];
    }
    return -1;
}

```

/**

查找两个子视图的共同父视图

思路:分别记录两个子视图的所有父视图并保存到数组中，然后倒序寻找,直至找到第一个不一样的父视图。

```

*/
- (void)findCommonParent:(UIView *)firstView
secondView:(UIView *)secondView {
    NSArray *firstParents = [self
getAllParentViewFor:firstView];
    NSArray *secondParents = [self
getAllParentViewFor:secondView];
    NSMutableArray *common = [NSMutableArray
array];
    NSInteger i = 0;

    while (i < MIN(firstParents.count,

```

```

secondParents.count)) {
    UIView *super1 =
firstParents[firstParents.count - i - 1];
    UIView *super2 =
secondParents[secondParents.count - i - 1];

    if (super1 == super2) {
        [common addObject:super1];
        i++;
    } else {
        break;
    }
}

- (NSArray *)getAllParentViewFor:(UIView *)view
{
    NSMutableArray *parents = [NSMutableArray
array];
    UIView *v = view;
    while ([v superview]) {
        [parents addObject:v];
        v = [v superview];
    }
    return [parents copy];
}

```

/**

求无序数组中的中位数

中位数：当数组个数 n 为奇数时，为 $(n + 1)/2$ ，即是最中间那个数字；当 n 为偶数时，为 $(n/2 + (n/2 + 1))/2$ ，即是中间两个数字的平均数

*/

```

- (void)findMiddleNum:(NSMutableArray *)nums {
    for (int i = 0; i < nums.count - 1; i++) {
        for (int j = i; j < nums.count; j++) {
            if (nums[j] < nums[i]) {
                [nums exchangeObjectAtIndex:j
withObjectAtIndex:i];
            }
        }
    }
}

```

```

        }
    }
    NSInteger num;
    if ((nums.count % 2) == 0) {
        num = [nums[nums.count / 2]
integerValue];
    } else {
        num = [nums[(nums.count + 1) / 2]
integerValue];
    }
}

```

/**

在一个字符串中找到第一个只出现一次的字符。如输入"abaccdeff"，输出'b'

字(char)是一个长度为8的数据类型，因此总共有256种可能。每个字母根据其ASCII码值作为数组下标对应数组中的一个数字。数组中存储的是每个字符出现的次数。

```

*/
+ (NSString *)findFirstOnceChar:(NSString *)str
{
    NSDictionary *map = [[NSDictionary alloc] init];

    for (int i = 0; i < str.length; i++) {
        int asciiCode = [str
characterAtIndex:i];
        if ([map objectForKey:@(asciiCode)]) {
            int count = [map
objectForKey:@(asciiCode)] intValue];
            [map setObject:@(count + 1)
forKey:@(asciiCode)];
        } else {
            [map setObject:@(1)
forKey:@(asciiCode)];
        }
    }

    NSString *chat;
    for (int i = 0; i < str.length; i++) {

```

```

        if ([[map objectForKey:@([str
characterAtIndex:i]]) intValue] == 1) {
            chat = [str
substringWithRange:NSMakeRange(i, 1)];
            break;
        }
    }

    NSLog(@"findFirstOnceChar ----- %@", chat);

    return chat;
}

/**
 输入为aaabbccc，输出为a3b2c3。不限语言。
 */
+ (NSString *)charCountFor:(NSString *)str {
    NSMutableDictionary *dict =
[NSMutableDictionary dictionary];
    NSMutableArray *keyTable = [NSMutableArray
array];

    for (int i = 0; i < str.length; i++) {
        NSString *s = [str
substringWithRange:NSMakeRange(i, 1)];
        if ([dict objectForKey:s]) {
            int count = [[dict objectForKey:s]
intValue];
            [dict setObject:@(count + 1)
forKey:s];
        } else {
            [dict setObject:@(1) forKey:s];
            [keyTable addObject:s];
        }
    }

    NSMutableString *result = [NSMutableString
string];

    for (NSString *key in keyTable) {
        [result appendFormat:@"%d%@", [dict

```

```
objectForKey:key]];
}
```

```
    NSLog(@"CharCountFor: %@", result);
```

```
    return result;
}
```

```
/**
```

```
    去掉字符串中不相邻的重复字符串    aabcad, aabcd
```

每次拿一个基准出来，将后面的进行比较，如果相等且位置不相邻，说明需要删除

```
*/
+ (void)removeStr:(NSString *)string {
    NSMutableString *str = [string mutableCopy];
```

```
    for (int i = 0; i < str.length - 1; i++) {
        unsigned char a = [str
characterAtIndex:i];
        for (int j = i + 1; j < str.length; j++)
        {
```

```
            unsigned char b = [str
characterAtIndex:j];
            //相等且位置不相邻，说明需要删除
            if (a == b && j != i + 1) {
                [str
deleteCharactersInRange:NSMakeRange(j, 1)];
            }
        }
    }
```

```
    NSLog(@"removeStr %@", str);
}
```

```
/**
```

```
    查找两个字符串的最大公共子串
```

思路：1. 找到比较短的一个字符串

2. 然后从长到短获取string1的子字符串,去string2

中找,如果包含,则返回该子字符串,否则再找

```
*/
- (NSString *)findMaxSubString:(NSString
*)string1 andString2:(NSString *)string2 {
    // 首先找到长度较小的字符串 保证string1<string2
    NSString *temp;
    if ([string1 length] > [string2 length]) {
        temp = string1;
        string1 = string2;
        string2 = temp;
    }

    long length1 = [string1 length];
    // 然后从长到短获取string1的子字符串,去string2中
    找,如果包含,则返回该子字符串,否则再找
    // location length
    NSRange range = NSRange(0, length1);

    // 如果不包含,并且range在合理范围内,也就是
    length >= 1 && location <= length - 1,
    // 进行调整range范围,继续查找
    while (![string2 containsString:[string1
    substringWithRange:range]] &&
        range.location <= length1 - 1 &&
    range.length >= 1) {
        // 判断下一步操作,
        if (range.location < length1 -
    range.length) {//不需要减length
            range.location++;
        } else {
            range.length--;
            range.location = 0;
        }
    }

    // 最后判断并返回子串
    if (range.length == 0) {
        NSLog(@"没有找到公共子字符串!");
        return nil;
    }
}
```

```

        } else {
            NSLog(@"找到了");
            return [string1
substringWithRange:range];
        }
    }

    struct Node {
        NSInteger data;
        struct Node *next;
        struct Node *prev;
    };

    - (void)invertLinkMap {
        struct Node *p = [self constructList];

        //反转后的链表头部
        struct Node *newH = NULL;

        //头插法
        while (p != NULL) {
            //记录下一个结点
            struct Node *temp = p->next;
            //当前结点的next指向新链表的头部
            p->next = newH;
            //更改新链表头部为当前结点
            newH = p;
            //移动p到下一个结点
            p = temp;
        }
    }

    - (void)reverseDouble {
        struct Node *pn = [self constructList];

        struct Node *q = NULL;
        struct Node *p = NULL;

        while(pn) {
            p = pn->next;

```

```

        pn->next = q;
        pn->prev = p;
        q = pn;

        pn = p;
    }

    // q 就是结果;
}

- (struct Node *)constructList {
    //头结点
    struct Node *head = NULL;
    //尾结点
    struct Node *cur = NULL;

    for (NSInteger i = 0; i < 10; i++) {
        struct Node *node = malloc(sizeof(struct
Node));

        node->data = i;

        //头结点为空，新结点即为头结点
        if (head == NULL) {
            head = node;
        } else {
            //当前结点的next为尾结点
            cur->next = node;
        }

        //设置当前结点为新结点
        cur = node;
    }

    return head;
}

- (void)printList:(struct Node *)head {
    struct Node *temp = head;

```

```

    printf("list is : ");

    while (temp != NULL) {
        printf("%zd ", temp->data);

        temp = temp->next;
    }

    printf("\n");
}

@end

/
*****
*****
二叉树
*/

@interface TreeNode ()
@property (nonatomic, assign) NSInteger value;
@property (nonatomic, strong) TreeNode *left;
@property (nonatomic, strong) TreeNode *right;
@property (nonatomic, assign) BOOL visited;
@end

@implementation TreeNode

- (instancetype)init
{
    self = [super init];
    if (self) {
        self.left = nil;
        self.right = nil;
        self.value = 0;
    }
    return self;
}

```

```

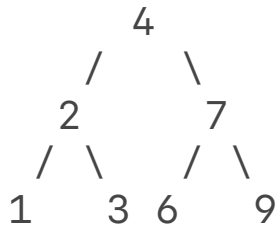
- (void)doIt {
    TreeNode *node = [self invertTree:[self
makeNodeTree]];
    [self printNode:node];
}

```

```

/*

```



```

*/

```

```

- (TreeNode *)makeNodeTree {
    TreeNode *oneNode = [[TreeNode alloc] init];
    oneNode.value = 1;
    TreeNode *threeNode = [[TreeNode alloc]
init];
    threeNode.value = 3;
    TreeNode *sixNode = [[TreeNode alloc] init];
    sixNode.value = 6;
    TreeNode *niceNode = [[TreeNode alloc]
init];
    niceNode.value = 9;

    TreeNode *twoNode = [[TreeNode alloc] init];
    twoNode.value = 2;
    twoNode.left = oneNode;
    twoNode.right = threeNode;
    TreeNode *sevenNode = [[TreeNode alloc]
init];
    sevenNode.value = 7;
    sevenNode.left = sixNode;
    sevenNode.right = niceNode;

    TreeNode *fourNode = [[TreeNode alloc]
init];
    fourNode.value = 4;
    fourNode.left = twoNode;

```

```

        fourNode.right = sevenNode;

        [self printNode:fourNode];

        return fourNode;
    }

- (void)printNode:(TreeNode *)node {
    if (!node.value) {
        return;
    }
    NSLog(@"-%ld", (long)node.value);
    [self printNode:node.left];
    [self printNode:node.right];
}

/**
    对称二叉树
*/
- (BOOL)isSymmetric:(TreeNode *)node {
    if (!node) {
        return YES;
    }
    return [self isMirror:node.left
node2:node.right];
}

- (BOOL)isMirror:(TreeNode *)node1 node2:
(TreeNode *)node2 {
    if (node1 == nil && node2 == nil) {
        return YES;
    }
    if (node1 == nil || node2 == nil) {
        return NO;
    }

    BOOL mirror = node1.value == node2.value &&
        [self isMirror:node1.left
node2:node2.right] &&
        [self isMirror:node1.right
node2:node2.left];

```

```

        return mirror;
    }

    /**
     相等的二叉树
    */
    - (BOOL)isSameTree:(TreeNode *)p q:(TreeNode *)q
    {
        if (p == nil && q == nil) {
            return YES;
        }
        else if (p != nil && q == nil) {
            return NO;
        }
        else if (p == nil && q != nil) {
            return NO;
        }
        else if (p != nil && q != nil && q.value !=
p.value) {
            return NO;
        }
        else {
            return [self isSameTree:p.left q:q.left]
&& [self isSameTree:p.right q:q.right];
        }
    }

    /**
     反转二叉树
    */
    - (TreeNode *)invertTree:(TreeNode *)root {
        //边界条件 递归结束或输入为空情况
        if(root == nil) {
            return root;
        }

        //交换左右节点
        TreeNode *temp = root.left;
        root.left = root.right;

```

```

        root.right = temp;

        //递归左右子树
        [self invertTree:root.left];
        [self invertTree:root.right];

        return root;
    }

```

```

/**
 是否是平衡二叉树

```

如果某二叉树中任意结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。

```

*/
- (BOOL)isBalanceTree:(TreeNode *)node {
    if (node == nil) {
        return YES;
    }

    NSInteger l = [self treeHeight:node.left];
    NSInteger r = [self treeHeight:node.right];

    if (ABS(l - r) > 1) {
        return NO;
    } else {
        return [self isBalanceTree:node.left] &&
        [self isBalanceTree:node.right];
    }
}

```

```

/**
 先序遍历：先访问根，再遍历左子树，再遍历右子树。典型的递归思想。

```

```

*/
- (void)preOrderTraverseTree:(TreeNode *)node {
    if (!node) {
        return;
    }
    NSLog(@"%ld", node.value);
}

```

```

        [self preOrderTraverseTree:node.left];
        [self preOrderTraverseTree:node.right];
    }

/**
    中序遍历，先遍历左子树，再访问根，再遍历右子树
 */
- (void)inOrderTraverseTree:(TreeNode *)node {
    if (!node) {
        return;
    }
    [self inOrderTraverseTree:node.left];
    NSLog(@"%ld", node.value);
    [self inOrderTraverseTree:node.right];
}

/**
    后续遍历
 */
- (void)postOrderTraverseTree:(TreeNode *)node {
    if (!node) {
        return;
    }
    [self inOrderTraverseTree:node.left];
    [self inOrderTraverseTree:node.right];
    NSLog(@"%ld", node.value);
}

/**
    广度优先
    用队列
 */
- (void)bfs:(TreeNode *)rootNode {
    if (!rootNode) {
        return;
    }
    //数组当成队列
    NSMutableArray *queueArray = [NSMutableArray
array];
    [queueArray addObject:rootNode]; //压入根节点

```

```

        while (queueArray.count > 0) {
            TreeNode *node = [queueArray
firstObject];
            //弹出最前面的节点，仿照队列先进先 出原则
            [queueArray removeObjectAtIndex:0];
            //压入左节点
            if (node.left) {
                [queueArray addObject:node.left];
            }
            //压入右节点
            if (node.right) {
                [queueArray addObject:node.right];
            }
        }
    }

/**
    深度优先
    stack<Node *> nodeStack;    //使用C++的STL 标准模
    板库
    nodeStack.push(root);
    Node *node;
    while(!nodeStack.empty()){
        node = nodeStack.top();
        cout<<node->data;//遍历根结点
        nodeStack.pop();
        if(node->rchild){
            nodeStack.push(node->rchild);    //先
将右子树压栈
        }
        if(node->lchild){
            nodeStack.push(node->lchild);    //再
将左子树压栈
        }
    }

    */

/**

```

二叉树的深度

```
*/
- (NSInteger)treeHeight:(TreeNode *)node {
    if (node == nil) {
        return 0;
    }

    NSInteger left = [self
treeHeight:node.left];
    NSInteger right = [self
treeHeight:node.right];

    return 1 + MAX(left, right);
}
```

@end

```
/
*****
*****
```

链表

```
*/
@interface LinkNode : NSObject
@property (nonatomic, strong) LinkNode *next;
@property (nonatomic, assign) NSInteger value;
@end
```

@implementation LinkNode

```
- (instancetype)init
{
    self = [super init];
    if (self) {
        self.value = 0;
        self.next = nil;
    }
    return self;
}
```

```
}
```

```
/**
```

```
单链表反转
```

```
头插法
```

```
*/
```

```
- (void)invertLinkMap:(LinkNode *)node {  
    LinkNode *newH;  
    while (node) {  
        LinkNode *temp = node.next;  
        node.next = newH;  
        newH = node;  
        node = temp;  
    }  
}
```

```
}
```

```
/**
```

```
如何检测一个链表中是否有环
```

用两个指针同时访问链表，其中一个的速度是另一个的两倍，如果它们变成相等了，那么这个链表就有环了

```
*/
```

```
- (BOOL)hasCycle:(LinkNode *)node {  
    LinkNode *slow = node;  
    LinkNode *fast = node;  
  
    while (fast && fast.next) {  
        slow = slow.next;  
        fast = fast.next.next;  
        if ([slow isEqual:fast]) {  
            return YES;  
        }  
    }  
    return NO;  
}
```

```
/**
```

```
删除链表中等于某数值的所有节点
```

需要前一个节点，和当前节点

1 2 3 4 5

```
*/
- (LinkNode *)removeNodeIn:(LinkNode *)head
target:(NSInteger)target {
    LinkNode *preNode;
    LinkNode *p = head;

    while (p) {
        // 相等
        if (p.value == target) {
            // 不是第一个节点
            if (preNode) {
                preNode.next = p.next;
            } else {
                head = p.next;
            }
        } else {
            preNode = p;
        }

        // 进行下一个节点比较
        p = p.next;
    }

    return head;
}
```

/**

给出一个链表和一个值X，要求将链表中所有小于X的值放到左边，所有大于等于X的值放到右边，并且原链表的节点顺序不能变

先处理左边（比x小的节点），然后再处理右边（比x大的节点），最后再把左右两边连起来

```
*/
- (LinkNode *)reSequenceLink:(LinkNode *)head
target:(NSInteger)target {
    LinkNode *left = [[LinkNode alloc] init];
```

```

    LinkNode *right = [[LinkNode alloc] init];
    LinkNode *node = head;

    while (node) {
        if (node.value < target) {
            left.next = node;
            left = node;
        } else {
            right.next = node;
            right = node;
        }
        node = node.next;
    }

    // 防止构成环
    right.next = nil;

    // 左右两部分相连
    left.next = right.next;

    return node;
}

/**
两个链表是否有交点

得到链表1和链表2的长度，分别记为 len1, len2，假设
len1 > len2
p1 指向链表1的头节点，p2 指向链表2的头节点
p1先移动 len1 - len2的距离
之后p1和p2同时移动，并比较两个指针指向的节点是否相同，
如果相同，则说明两个链表相交，返回该节点。如果不相同，继续
移动，直到链表结尾，说明两个链表不相交。
*/
- (LinkNode *)getIntersectionNode:(LinkNode
*)headA node:(LinkNode *)headB {
    NSInteger aLength = [self
lengthOfLink:headA];
    NSInteger bLength = [self
lengthOfLink:headB];

```

```

    NSInteger dif = aLength - bLength;

    LinkNode *p1 = headA;
    LinkNode *p2 = headB;

    for (int i = 0; i < ABS(dif); i++) {
        // a long
        if (dif > 0) {
            p1 = p1.next;
        } else {
            p2 = p2.next;
        }
    }

    while (p1 && p2) {
        if ([p1 isEqual:p2]) {
            return p2;
        }
        p1 = p1.next;
        p2 = p2.next;
    }

    return nil;
}

/**
合并两个有序链表

递归调用
*/
- (LinkNode *)mergeTwoLists:(LinkNode *)node1
node2:(LinkNode*)node2 {
    if (node1 == nil) {
        return node2;
    }
    if (node2 == nil) {
        return node1;
    }

    LinkNode *mergedNode;

```

```

        // 递归调用
        if (node1.value < node2.value) {
            mergedNode = node1;
            mergedNode.next = [self
mergeTwoLists:mergedNode.next node2:node2];
        } else {
            mergedNode = node2;
            mergedNode.next = [self
mergeTwoLists:node2 node2:mergedNode.next];
        }

        return mergedNode;
    }

/**
 链表长度
*/
- (NSInteger)lengthOfLink:(LinkNode *)node {
    if (node == nil) {
        return 0;
    }

    NSInteger count = 0;
    while (node) {
        count++;
        node = node.next;
    }
    return count;
}

@end

```