说明

一、运行环境

1. 语言: c++ 11

2. 编译器(g++)版本

```
~ g++ -v
Configured with: --prefix=/Applications/Xcode.app/Contents/Developer/usr --with-gxx-include-dir=/usr/include/c++/4.2.1
Apple LLVM version 9.0.0 (clang-900.0.39.2)
Target: x86_64-apple-darwin17.4.0
```

3. 编译命令

```
g++% -g -o %< -Wall -std=c++11
```

二、第一题

> 设计算法，将存有n(n>0)个数的数组A中的元素A[0]至A[n-1]循环右 移k(k>0)位，要求只允许使用一个元素大小的附加存储，元素移 动或交换次数为O(n)。

1. 测试

测试数组

```
int arr[] = {1, 2, 3, 4, 5, 6, 7, 8};
```

测试`k=3`的输出结果

```
Result
6 7 8 1 2 3 4 5
```

测试测试`k=6`的输出结果

```
Result
3 4 5 6 7 8 1 2
```

2. 算法说明

设数组为a，长度为len。

首先考虑两个特殊情况

1. `k = 0`，不用移动
2. `k >= len`时，显然移动k位的结果和移动`k % len`位的结果相同，所以

```
if (k >= len) k %= len;
```

接着考虑一般情况 假设k=3吧，数组从0开始，长度是5，这样方便描述。 我们要做的是把a[0]移到a[3]，a[3]移到a[1]，a[1]移到a[4]，a[4]移到a[2]，a[2]移到a[0]。总得来说，就是把a[i]移到a[(i + k) % len]上。什么时候结束呢？我们之前说的这个例子是从0开始，到0结束。所以我们很显然的发现，如果重新回到起点，就结束了。即

```
temp = a[i];
j = i;
do {
    j = (j + k) % len;
    swap(a[j], temp);
} while (j != i);
```

好了，考虑另一个情况，k=3，长度是6。这样有什么区别呢？我们来看一看。

1. 把a[0]移到a[3]，a[3]移动到a[0]
   诶，怎么就结束了？那么我们还需要进行下面两个步骤
2. 把a[1]移到a[4]，a[4]移动到a[1]
3. 把a[2]移到a[5]，a[5]移动到a[2]
   这次的话，我们一共进行了3次上述的循环！我们暂且管这种循环叫做链式移动。

那么问题来了，我们到底要进行多少次链式移动呢？答案是`gcd(len, k)`次，其中`gcd`是最大公约数。 证明如下： 从i开始，每次i = (i + k) % len，直到回到起点，假设一个链式移动进行n次，即：

$$i + nk \equiv i \pmod{len}$$

写成代码就是`(i + nk) % len == i`，显然

$$len \mid nk$$

我们可以写成

$$\exists\, m \in N,\ \ len\, m = nk$$

我们引入三个新的变量，d, p, q，定义为

$$\begin{gathered} d = gcd(len, k) \\ k = dp \\ len = dq \end{gathered}$$

很显然的一个性质是$gcd(p, q) = 1$。现在将这些变量带入 $len\, m = nk$，得到

$$dq\, m = n\, dp$$

化简

$$qm = np$$
$$n = m\frac{q}{p}$$

因为n是最小的满足以上条件的整数，p和q互质，所以显然取$m = p$才能够得到最小的n，即

$$n = m\frac{q}{p} = q$$

根据q的定义$len = dq$我们可以推出

$$q = \frac{len}{d} = \frac{len}{gcd(len, k)}$$

所以得

$$n = \frac{len}{gcd(k, len)}$$

所以每次链式移动能够移动n个数，一共有len个数需要移动，所以一共需要

$$\frac{len}{n} = gcd(k, len)$$

次链式移动。用代码实现便是：

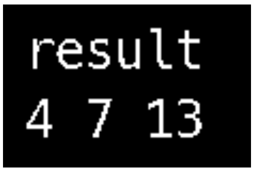
```
int temp; // 只允许使用一个元素大小的附加存储
for (int i = 0, j; i < gcd(k, len); i ++) {
    temp = a[i];
    j = i;
    do {
        j = (j + k) % len;
        swap(a[j], temp);
    } while (j != i);
}
```

三、第二题

> 用循环队列编写求k阶斐波那契序列中前n+1项($f_1, f_2, ..., f_n$)的算法，要求满足$f_n \le max$，而$f_{n+1} > max, max$为某个约定的常数，注意：本题所用循环队列的容量为k，算法结束时，留在队列中的元 素为所求k阶斐波那契序列中的最后k项。

1. 测试

`k = 3, MAX = 20`时，结果如下



2. 算法说明

k阶斐波那契数列的定义

$$
\begin{aligned}
f_1 = \cdots = f_{k-1} &= 0 \\
f_k &= 1 \\
f_m &= f_{m-1} + \cdots + f_{m-k}
\end{aligned}
$$

简单的推导后得到

$$
\begin{aligned}
\because f_m &= f_{m-1} + \cdots + f_{m-k} \\
f_{m-1} &= f_{m-2} + \cdots + f_{m-k-1} \\
\therefore f_m &= 2f_{m-2} - f_{m-k-1}
\end{aligned}
$$

但是由于**题目严格要求了空间复杂度**，所以我们只能选择牺牲时间复杂度，不存储$而用加和的方式来计算。

四、第三题

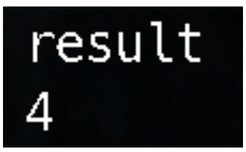
> (1)创建二叉树的二叉链表存储表示，(2) 利用循环队列设计非递归算法实现二叉树的层次遍历，(3)设计非递归算法求二叉树的宽度(即:每层结点数的最大值)。

1. 测试

因为题目未做明确要求，我们就随便建一棵树

```
      A
    /   \
   B     E
  / \   / \
 C   D F   G
```

输出宽度



2. 算法说明

`int bfs(Node * head)`：层次遍历，返回宽度

`Node * build_test_tree()`: 建一个测试用的树

五、第四题

> 若矩阵$Am \times n$中的某个元素a_{ij}同时是第i行和第j列的最小值，称为马鞍点。求二维矩阵$Am \times n$中的马鞍点。

测试

随便建一个矩阵A

$$A = \begin{bmatrix} 1 & 2 & 3 \\ 4 & 5 & 2 \end{bmatrix}$$

输出马鞍点位置

```
pos (0, 0) is a saddle point
pos (1, 2) is a saddle point
```

算法说明

最坏时间复杂度`O(nm)`

首先初始化，获得第`j`列的最小值`col_min[j]`，时间复杂度O(mn)。

```
for (int j = 0; j < A.m; j ++) {                            // init
    col_min[j] = inf;
    for (int i = 0; i < A.n; i ++)
        col_min[j] = min(col_min[j], A.a[i][j]);
}
```

然后找每一行的最小值，记录位置`pos`，和`col_min[pos]`比较，如果相等显然就是马鞍点了。若每一列每一个值都相等，最坏时间复杂度`O(nm)`。