作业6

- 课本48页， 2.3， 2.4， 2.5，2.6(2)， 2.7(2)(3)

2.3

2.3 双 *Hanoi* 塔问题是 *Hanoi* 塔问题的一种推广，与 *Hanoi* 塔的不同点在于： $2n$ 个圆盘，分成大小不同的 n 对，每对圆盘完全相同. 初始，这些圆盘按照从大到小的次序从下到上放在 A 柱上，最终要把它们全部移到 C 柱，移动的规则与 *Hanoi* 塔相同.

(1) 设计一个移动的算法并给出伪码描述.

(2) 计算你的算法所需要的移动次数.

2.3

2.3 (1)算法设计思想：分治策略. 先递归地将上面的 $2(n\text{-}1)$ 个盘子从 A 柱移到 B 柱；用2次移动将最大的2个盘子从 A 柱移到 C 柱；递归地将 B 柱的 $2(n\text{-}1)$ 个盘子从 B 柱移到 C 柱. 伪码描述如下：

```
BiHanoi(A,C,n)          //从A到C移动2n只盘子
1.  if n == 1 then
2.        BiMove(A,C)       //从A到C移动2只盘子
3.   else
4.        BiHanoi(A,B,n-1)
5.        BiMove(A,C)
6.        BiHanoi(B,C,n-1)
7.  end
```

(2) 设 $2n$ 个圆盘的移动次数是 $T(n)$, 则第行和第4行的递归调用的子问题规模是 n-1, 第3行是2次移动, 于是有

$$\begin{cases} T(n) = 2T(n-1) + 2 \\ T(1) = 2 \end{cases}$$

解得 $T(n) = 2^{n+1} - 2$

2.4

2.4 给定含有 n 个不同的数的数组 $L=<x_1,x_2,\ldots,x_n>$，如果 L 中存在 x_i，使得 $x_1<x_2<\ldots<x_{i-1}<x_i>x_{i+1}>\ldots>x_n$，则称 L 是单峰的, 并称 x 是 L 的"峰顶". 假设 L是单峰的, 设计一个算法找到 L 的峰顶.

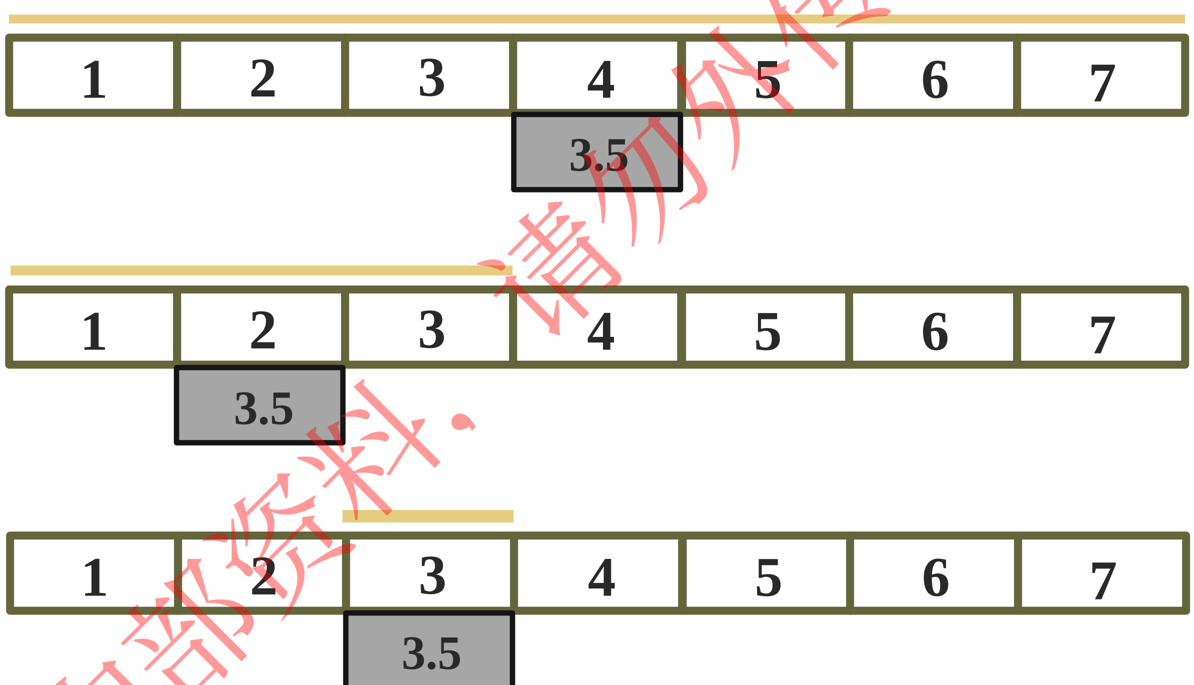
因为 L 中存在峰顶元素, 因此 $|L|\geq 3$. 使用二分查找算法. 如元素数等于3, 则$L[2]$是峰顶元素. 当元素数 n 大于3时, 令 $k=\lfloor n/2 \rfloor$，比较 $L[k]$ 与它左边和右边相邻的项. 如果 $L[k]>L[k-1]$ 且 $L[k]>L[k+1]$，则 $L[k]$ 为峰顶元素；否则，如果$L[k-1]>L[k]>L[k+1]$，则继续搜索 $L[1\ldots k-1]$ 的范围；如果 $L[k-1]<L[k]<L[k+1]$，则继续搜索 $L[k+1\ldots n]$ 的范围. 每比较2次，搜索范围减半，直到元素数小于等于3停止递归调用. 时间复杂度函数为:

$$\begin{cases} T(n)=T(\dfrac{n}{2})+2 \\ T(1)=c, \quad c\text{为某个常数} \end{cases}$$

根据主定理，$T(n)=O(\log n)$.

二分查找法

二分检索运行实例：$x = 3.5$



第1次
3.5<4

第2次
3.5>2

第3次
3.5>3

$X = 2.8$

1.是否正确？
2.如何改进？
3.如何验证改进是正确的？

```
l ← 1;   r ← n
While l ≤ r do
        m ← ⌊(l+r)/2⌋;
        if  L[m-1] < L[m] > L[m+1]  then
              return L[m];
        elif L[m-1] < L[m] < L[m+1]  then
              l ← m+1;
        else           // L[m-1] > L[m] > L[m+1]
              r ← m-1;
        end
end
```

特例： 12354
特例： 15432

改进一：

特殊情况特殊处理：$l\text{-}r>1$

改进二：

$l \leftarrow m$

$r \leftarrow m$

验证：长度为3，4，5；之后都是重复

```
l ← 1;  r ← n
While l ≤ r do
        m ← ⌊(l+r)/2⌋;
        if  L[m-1] < L[m] > L[m+1]  then
            return L[m];
        elif L[m-1] < L[m] < L[m+1]  then
            l ← m;
        else            // L[m-1] > L[m] > L[m+1]
            r ← m;
        end
end
```

改进一：
特殊情况特殊处理：l-r>1

改进二：
$l \leftarrow m$
$r \leftarrow m$

改进三（严中圣）：
初始 $l \leftarrow 2$

验证：*长度为3，4，5；之后都是重复*

```
l ← 2;  r ← n
While l ≤ r do
        m ← ⌊(l+r)/2⌋;
        if  L[m-1] < L[m] > L[m+1]  then
            return  L[m];
        elif  L[m-1] < L[m] < L[m+1]   then
              l ← m+1;
        else              // L[m-1] > L[m] > L[m+1]
              r ← m-1;
        end
end
```

改进一：

特殊情况特殊处理：*l*-*r*>1

改进二：

$l \leftarrow m$

$r \leftarrow m$

改进三（严中圣）：

初始 $l \leftarrow 2$

改进四（罗涛）：

单边判断

验证：长度为3，4，5；之后都是重复

```
faction  Champion(L)
l ←1;  r ←n
While l <  r  do
    m ←⌊(l+r)/2⌋;
    if  L[m] < L[m+1]  then
        l ←m+1 ;
    elif  L[m] >L[m+1]  then
        r←m ;
    end
end
return  l
```

改进四：单边判断

　　广义单峰问题：数列严格递增、严格递减或书上所指单峰。严格递增（峰顶是尾）或者严格递减（峰顶是首）。
显然，如果算法能解决"广义单峰"，那书上的问题也能解决。

idea：利用分治算法，最简子问题只对两个数或三个数进行比较，l 找到满足条件的数，只需要直接返回 l 即可。

算法思想：对于一个满足上述定义的序列：
如果中间的数L[mid]，则返回；
如果中间的数L[mid]比下一个小，那L[mid]绝对不可能是峰顶,峰顶只有可能出现在右边，并且右边序列任满足定义；
如果 L[mid]比下一个大，那峰顶绝不可能在右边，只可能存在于左边包含L[mid]的序列，此时，该序列也满足定义。

改进四：单边判断

递归表示

```
Function champion(l,r):
if(l==r) then
        return l
end
        m ← floor((l+r)/2)
if (L[m] < L[m+1])  then
        return champion(m+1,r)
else
        return champiton(l,m)
end
end
```

迭代表示

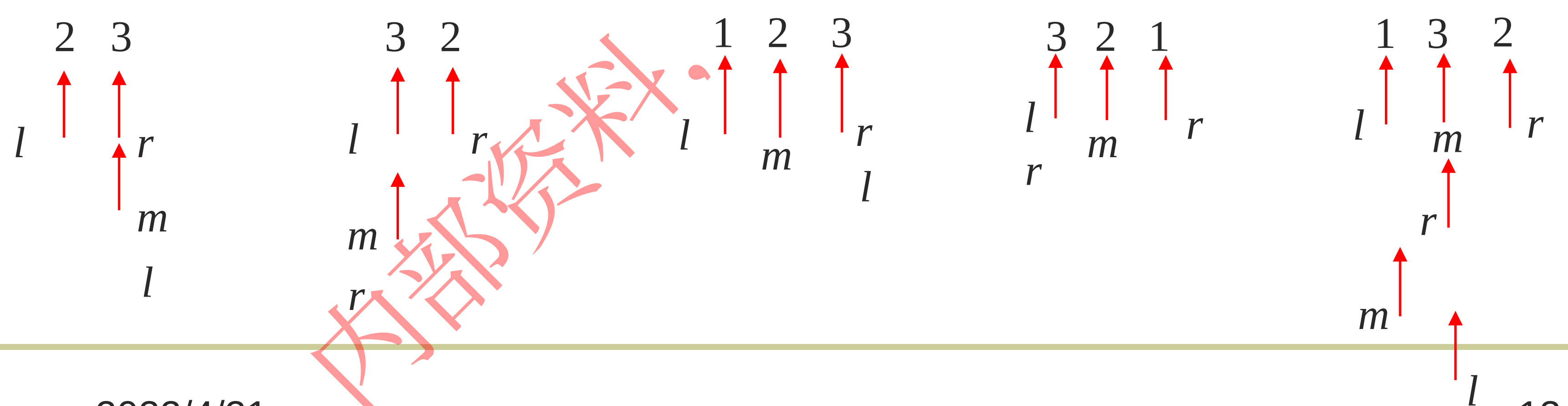
```
faction  Champion(L)
l ←1;  r ←n
While l < r do
        m ←⌊(l+r)/2⌋;
        if  L[m] < L[m+1]  then
            l ←m+1 ;
        else
            r←m ;
        end
end
return  l
```

正确性

- 最小规模问题只有两种，规模为2，或3（更大规模的都会归约到这两种情况）。
- 结合上诉定义，规模为2只能是递增或递减序列，举例验证，均可以找到封顶；
- 规模为3可能是递增或递减序列，也可能是书上严格定义的单峰，也是均可以找到峰顶。
- 至此“我”认为，不需要再验证更大规模的问题

```
faction  Champion(L)
l ←1;  r ←n
While  l <  r  do
        m ←⌊(l+r)/2⌋;
        if  L[m] < L[m+1]  then
            l ←m+1 ;
        elif  L[m] >L[m+1]  then
            r←m ;
        end
end
return  l
```

最简子问题只有两种情况，规模为2，或3（更大规模的都会规约到这两种情况）。



2.4　使用蛮力法，直接查找？

2.4 解:

```
Search(L)
s ← L[1];
for i ← 2 to n do
    if L[i] > s then
        s ← L[i];
    else
        break;
    end
end
return s;
```

2.5

2.5　设 A 是 **n 个不同的数排好序**的数组，给定数 L 和 U，$L<U$，设计一个算法找到 A 中满足 $L<x<U$ 的所有的数 x.

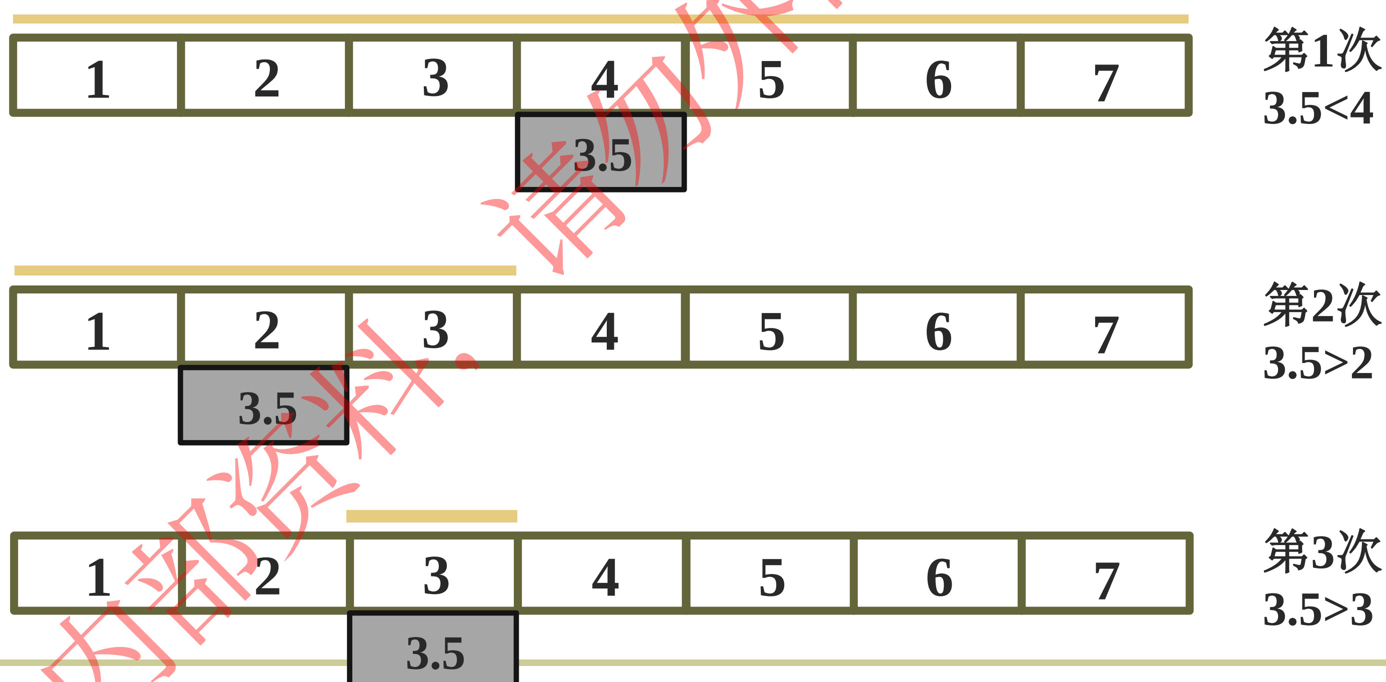
```
输入：数组A,数L,U
输出：A中满足L<x<U的所有x,否则输出0

if L ≥ A[n] or U ≤ A[1] then
    return 0;
else if L < A[1] and U > A[n] then
    return A;
else
    i ← 1;
    j ← n;
    repeat i ← i+1 until A[i] ≥ L;
    repeat j ← j-1 until A[j] ≤ U;
    return [A[i], A[i+1], …, A[j]];
end.
```

```
2.5 输入：排好序的数组A，给定的数L,U且L<U
输出：A中所有满足L<x<U的x下标i.
if A[n]≥U then
    return 0;
end
while i≤n do    //找到第一个大于L的x.
    if A[i]≤L then
        i←i+1
    else
        break;
    end
end
if i==n+1 then
    return 0;    //数组A的数都比L小.
end
add i to B;    //数组B存储下标.
while (j←i+1)<n then
    if A[j]<U then
        j←j+1;
        add j to B;    //将j添加到数组B中.
    else
        break;
    end
end
return B;
```

二分查找法

$A[j_1 - 1] \le L < A[j_1]$

- 第一步：在 *A* 中寻找第一个大于 *L* 的数；
- 1.蛮力法，从前向后逐个寻找；
- 2.二分查找法



二分查找法

- 第一步：在 A 中寻找第一个大于 L 的数；
- 1.蛮力法，从前向后逐个寻找；
- 2.二分查找法

1	2	3	4
	1.1 <2		

第1次
1.1<2

第1次
3.4>2

1	2	3	4
	3.4 >2		

1
1.1 >1

第2次
1.1>1

第2次
3.4>3

3	4
3.4 >3	

$l \leftarrow m+1$

第3次
3.4<4

4
3.4 <4

$r \leftarrow m-1$

截止条件： **A[*m*]==*x*** ； l>r(l=r+1)

A[m]=A[r]<x<A[l]　　A[r]<x<A[l]=A[m]

2.5

```
Searchbigger(A,x)
l←1; r←n;
while l≤r do
    m←⌊(l+r)/2⌋;
    if A[m]==x  then
        break;      // 恰好等于中位元素
    elif A[m]>x  then
        r ← m–1;
    else
        l←m+1;
    end
end
if A[m]>x then
    return m;
elif  m<n then
    return m+1;
else
    return 0;
end
```

A[m]==x； A[m]>x; A[m]<x

```
Searchbigger(A,x)
l←1; r←n;
if A[n] ≥ x then
    return 0;
end
while l ≤ r do
    m←⌊(l+r)/2⌋;
    if A[m]==x  then
        break;      // 恰好等于中位元素
    elif A[m]>x  then
        r ← m–1;
    else
        l←m+1;
    end
end
if A[m]>x then
    return m;
else
    return m+1;
end
```

2.5

```
Searchbigger(A,x)
    l←1; r←n;
    if A[n] ≥ x then
        return 0;
    end
    while l< r do
        m←⌊(l+r)/2⌋;
        if  A[m]==x  then
            return m+1;      // 恰好等于中位元素
        elif  A[m]>x  then
            r ← m–1;
        else
            l←m+1;
        end
    end                  l=r
    if A[l]>x then
        return l;
    else
        return l +1;
    end
```

```
Searchbigger(A,x)
    l←1; r←n;
    if A[n] ≥ x then
        return 0;
    end
    while l< r do
        m←⌊(l+r)/2⌋;
        if A[m] ≤ x  then
            l←m+1;
        else               //A[m]>x
            r ← m–1;
        end
    end           l≥r : l=r+1; l=r
    return l;
```

1	2	3	4
	2	2.4	3.4

类似查找第一个和最后一个等于 x 的值

- 1.找到第一个符合要求的数后，继续依次向后寻找;

```
j1 ← Searchbigger(A,L)
if j1 ==0 or A[j1]≥U then
    return 0
elif j1 ==n then
    return j1 , j1
else
    for j2 ← j1 +1 to n do
        if L[j2] ≥U then
            j2 ← j2 -1;
            break;
        end
    end
end
return j1, j2
```

类似查找第一个和最后一个等于 x 的值

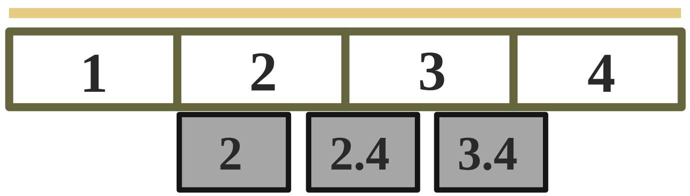
- 1.找到第一个符合要求的数后，继续依次向后寻找;
- 2.新的二分查找函数;

```
j1 ← Searchbigger(A,L)
if j1 ==0 or A[j1]≥U then
    return 0
elif j1 ==n then
    return j1 , j1
else
    for j2 ← j1 +1 to n do
        if L[j2] ≥U then
            j2 ← j2 -1;
            break;
        end
    end
end
return j1, j2
```

$A[j_2] < U \le A[j_2 + 1]$

二分查找函数：寻找第一个小于U的数

```
Searchlower(A,x)
l←1; r←n;
if A[1] ≥ x then
    return 0;
end
while l< r do
    m←⌊(l+r)/2⌋;
    if A[m]==x then
        return m-1;
    elif A[m]>x then
        r ← m−1;
    else            //A[m]<x
        l←m+1;
    end
end    截止条件：l=r
if A[l]<x then
    return l;
else
    return l -1;
end
```




```
j1 ← Searchbigger(A,L)
if j1 ==0 or A[j1]≥U then
    return 0
elif j1 ==n then
    return j1 , j1
else
    j2 ← Searchlower(A,U)
end
return j1, j2
```

```
Searchlower(A,x)
l←1; r←n;
if A[1] ≥ x then
    return 0;
end
while l< r do
    m←⌊(l+r)/2⌋;
    if A[m] ≥ x then
        r ← m−1;
    else          //A[m]<x
        l←m+1;
    end
end    l≥r : l=r+1; l=r
if A[r]<x then
    return r;
else
    return r -1;
end
```

3.再次调用*Searchbigger*

if $A[n] \le L$ **or** $A[1] \ge U$ **then**
  return 0
end
$j_1 \leftarrow Searchbigger(A,L)$
if $A[j_1] \ge U$ **then**
  return 0
elif $j_1 == n$ **then**
  return j_1, j_1 $L<A[j_1]<U$; $j_1 \mathrel{!=} n$;
else
  $j_2 \leftarrow Searchbigger(A[j_1{:}n], U)$ $A[j_2 - 1] \le U < A[j_2]$
  if $A[j_2 - 1] < U$ **then**
    $j_2 \leftarrow j_2 - 1$;
  else
    $j_2 \leftarrow j_2 - 2$;
  end
end
return j_1, j_2;

2.6

2.6 设 M 是一个 n 行 n 列的0-1矩阵，每行的1都排在0的前面，寻找到 M 中含有1最多的行.

(2) 请设计一个最坏情况下 $O(n)$ 时间的算法，并给出算法伪码

$$\begin{pmatrix} 1 & 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 0 & 0 & 0 \\ 1 & 1 & 1 & 1 & 1 & 0 \\ 1 & 1 & 1 & 1 & 0 & 0 \end{pmatrix}$$

```
i←1 ;  j←1 ;  max ←1 ;
while   i ≤ n  and  j ≤ n  do
        if  M[i,j] == 1  then
                j ←j+1 ;
                max ←i ;
        else
                i ←i+1 ;
        end
end
return   max
```

2.7

2.7 设 A 是含有 n 个元素的数组，如果元素 x 在 A 中出现的次数大于 $n/2$，则称 x 是 A 的主元素.

(2)对于已排好序的数组,能否设计一个 $O(n)$ 时间的算法，判断 A 中是否存在主元素?

(3)如果 A 中元素只能进行“是否相等”的测试，但是不能排序，设计一个$O(n)$ 时间的算法判断 A 中是否存在主元素.

命题2.2　A 中的主元素一定是中位数.

证　按照从小到大排序 A 的元素．若存在主元素 x，那么 x 应该至少占有连续的 $\lfloor n/2 \rfloor+1$ 个位置．如果其中不包含中位数的位置在内，那么这些连续位置只能分布在中位数的单侧．中位数任何一侧的元素数都不超过 $\lfloor n/2 \rfloor$．这与主元素的定义矛盾.

找中位数，计数

2.7(2)假设已经排好序

```
i = ⌈n/2⌉;
x ← A[i];
for j ← 1 to n do
    if A[j]==x then
        count ← count+1;
    end
end
if count > n/2 then
    major ← x;
else
    major ← 0;
end
return major;
```

也可以分别向左、向右统计并计数

2.7(3)

- 采用何种算法?
- 芯片测试

- 如何进行具体设计?
- 1.算法的主体逻辑
- 2.最小规模子问题
- 3.细节问题:轮空处理

芯片测试方法

其设计思想如下:

算法 ***Delete(A)***

1. 如果 $|A| \le 1$ 则结束;否则将 A 的元素两两分组.
2. 每组内两个元素进行比较;如果相等则把1个放到 B 中;否则全部淘汰.
3. $A \leftarrow B$ 并转1.运行上述 ***Delete*** 淘汰过程,如果最后没有元素留下,那么没有主元素. 如果剩下1个元素,接着检查该元素在 A 中出现的次数.若次数大于 $n/2$,则是主元素;否则没有主元素.

命题2.3 假设 n 是偶数,经过一次 ***Delete(A)*** 的淘汰过程, A 具有以下性质.

性质1: 如果 A 中含有主元素,淘汰后的主元素数仍旧多于非主元素数.

性质2: A 的元素至多为原来的一半.

证 因为 n 是偶数,在分组中没有轮空的元素.考虑分组的3种类型.

① 含有两个主元素:有 a 个组 （一定相同）

② 含有一个主元素和一个非主元素:有 b 个组 （一定不同）

③ 含有两个非主元素:有 c 个组 （有可能相同）

淘汰前的主元素数为 $2a+b$,非主元素数为 $2c+b$. 因为 $2a+b>2c+b$,从而有 $a>c$;

根据 ***Delete*** 第2行的淘汰规则,在1次淘汰后,主元素数为 a ,非主元素数至多为 c ;

由于 $a>c$,淘汰后主元素数仍旧多于非主元素数.

此外,由于每组至少淘汰1个元素,所以剩下的元素数至多是原来的一半.

轮空元素的处理

当 n 是奇数时，如果被轮空的元素不是主元素时，有可能淘汰后的主元素数等于非主元素数. 比如数组元素为1,1,1,2,3, 分组是{1,1},{1,2},3. 被轮空，经过1轮 ***Delete*** 淘汰后，剩下的是1,3. 可以用下述办法解决这个问题：

检查轮空元素在 A 中出现的次数，如果该元素是主元素，则算法结束；如果不是，则在下一轮 ***Delete*** 过程开始之前先把这个元素淘汰掉，从而保证每次进入 ***Delete*** 过程的剩余数组都满足“主元素数大于非主元素数”的性质.

******************（分组淘汰前，一次蛮力算法）************************

由**命题2.3**可以断定，如果 A 有主元素，它一定会在所有的 ***Delete*** 过程结束后留下来. 然而：“留下来”是主元素的必要条件，但不是充分条件，即可能有非主元素也会留下来.

反例：输入是1,2,3,4,5,6,7,7，没有主元素. 如果第一次分组是{1,2},{3,4 },{5,6},{7,7}，那么一轮 ***Delete*** 淘汰后只剩下7，但7不是主元素.

因此，必须对淘汰后剩下的元素进行验证，即检查它在原始数组A中出现的次数.

能否根据分组淘汰后再处理轮空元素？

```
k ← n; B ← A;
while  k >1  do
    if  k 为奇数 then
        查询B[k]在原数组A中出现次数      //也可任取B中的一个元素
        if  次数 > n/2  then
            return B[k];
        else
            删除B[k]， k ← k-1 ;
        end
    else
        将B中元素分成k/2 组;
        for  i ← 1  to  k/2  do
            if  2个元素同  then
                任取1片放入B中，另一片删掉；
            else
                B中2片同时去掉;
            end
        end
        k ←B中元素个数;
end
if  k ==1   then
    A中查询出现B[1]出现次数;
    if  次数 > n/2  then
        return B[1];
    end
end
return 0
```

方法二：栈淘汰方法

设计思想:设一个栈 S，大小为 $\left\lceil \frac{n}{2} \right\rceil$. 顺序对 A 中的元素x进行以下处理:

1. 如果栈为空，则 x 进栈.
2. 如果栈不空，则比较 x 与栈顶元素. 如果相等，则 x 进栈；否则 x 与栈顶元素一起淘汰.

命题2.4 在上述栈操作后，栈具有下述性质.

性质1: 如果栈不空,那么其中的元素都相等.

性质2: 如果 A 中有主元素,那么它一定留在栈中.

证 假设栈内剩余的元素不等，那么一定存在两个相邻的元素不等，比如 x 和 y，y 在 x 的上面. Y 进栈时，x 为当时的栈顶元素，这与栈的操作规则2矛盾.

假设 x 为 A 中的主元素，x 的个数一定大于 $n/2$. 如果 x 在进栈时被淘汰，根据规则2，当时栈顶元素不是主元素，而这个元素也将同时被淘汰. 于是，要淘汰所有的主元素，至少需要多于 $n/2$ 个非主元素，这与 A 中元素总数等于 n 矛盾.

与前面的情况类似，主元素一定留在栈里，但留在栈里的有可能是非主元素. 因此:

在 A 中元素全部处理完成后，如果栈为空，则没有主元素;

如果栈内剩有元素，那么检查栈顶元素在 A 中出现的次数. 次数大于 $n/2$ 的是主元素;

否则没有主元素;

该算法的时间是 $W(n)=O(n)+O(n)=O(n)$.

2.7(3)

```
S ← 栈, count ← 0;
for i ← 0 to n do
    if S. top == -1 then      //栈为空
        push(A[i]);           //将第i个数放入栈并且top+1
    else
        if A[i] == seek(S)   //第i个数等于栈顶元素
            push (A[i]);
        else
            pop(S);           //将栈顶元素取出并且top-1
        end
    end
end
if S. top == -1 then
    return "不存在主元素";
else
    for i ← 0 to n do
        if A[i] == seek(S) then
            count++;
        end
    end
end
if count > n/2 then
    return pop(s)
else
    return "不存在主元素"
end
```

111111122222
12121212121
1234567111111