231275036 朱晗 作业3

要求

任选 20 题

- 5.2, 5.4, 5.5, 5.6, 5.7, 5.9, 5.10
- 6.2, 6.3, 6.4, 6.5, 6.8, 6.14
- 15.1, 15.3
- 16.1, 16.4
- 18.1, 18.3, 18.5
- 19.1, 19.3, 19.4，19.5, 19.6，19.10

题目

Chapter 5 （6）

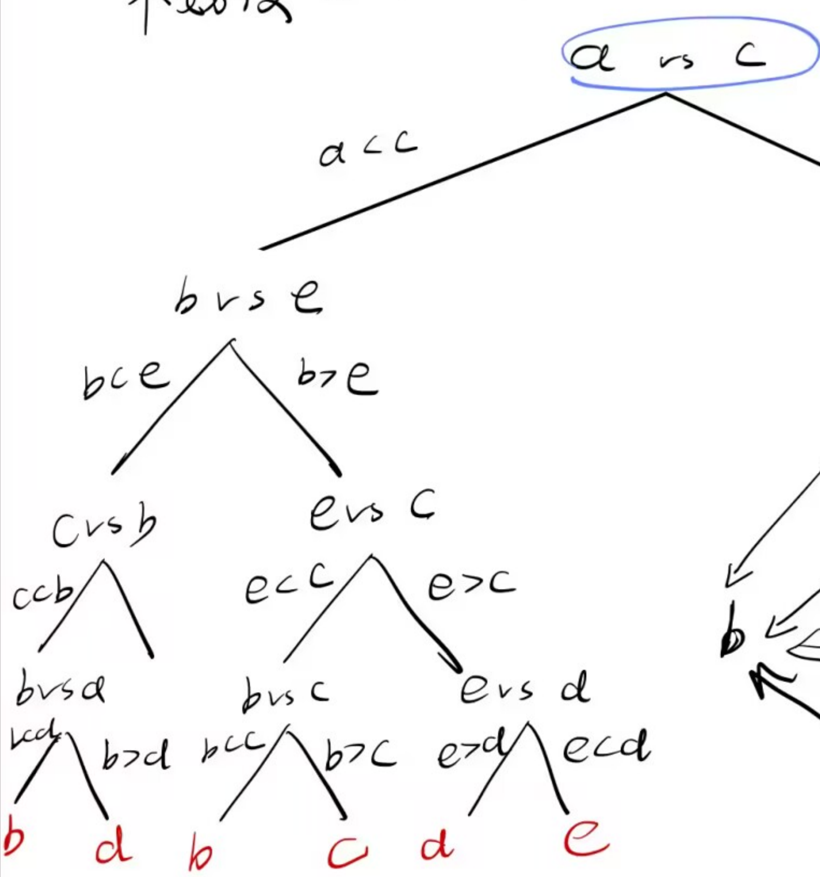
5.2（6 次比较找 5 个元素中位值）

Initial: 2 comparisons.

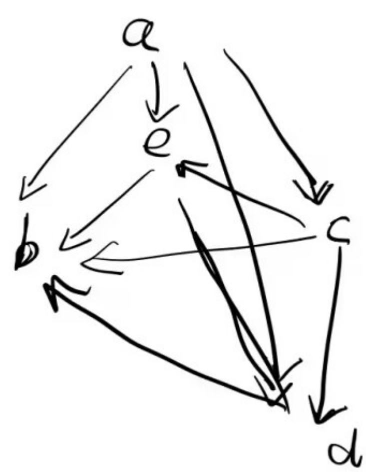
a vs b.   c vs d.

不妨设 a<b and c<d.

Insight:
找中位数:
是偏序关系



完全类似.



如图所示，有 中位数 需要找到偏序关系这个 Insight，容易做出。图中初始先进行了两次比较得到 `a<b && c<d` 然后决策树高度最深为 4，所以是可以最多 6 次比较得到答案的。

5.4

一个算法只用比较来确定阶为 i 的元素，证明：无需额外的比较，也能找到比该元素小 i-1 个元素和比该元素大的 n-i 个元素

`proof`

一个算法如果能确定阶为 i 的元素，其必然建立了这样一个 偏序 关系，即所有元素与阶为 i 的大小关系可知。类似于上一题我画出的偏序图，所以不需要额外的比较，就可以找到以上元素。

5.5

我们采取这样的分治的思想：

- 找到中位数
- 以中位数作为 pivot 做 partialize
- 递归找，直到阶是我们所求

```python
def A_alg(data[])#已有黑盒算法，实现不展示

def B_alg(data[],int k):#k是我们找的阶数
    int mid=A_alg(data)
    Partial(data,mid) #以mid做分界，左边比mid小，右边比mid大
    if(k<len(data)/2):#在左侧找，只需要检查左侧元素。
        return B_alg(data,k)
    else:#在右侧找，找k-len(data)/2阶
        return B_alg(data,k-len(data)/2)
```

证明时间复杂度：

该算法满足 `T(n)=O(n)+T(n/2)`，即子问题规模每次变成一般，且子问题个数只有一个（根据比较 k 和中位数阶数的关系来选择）。由主定理容易知道其在线性时间即 $O(n)$ 内做完

5.7

给定一个 n 个不同整数的集合 `S`，用 `M` 表示 `S` 的中位数，请设计算法找到和 `S` 中和 `M` 大小最接近的 k 个数（`k<<n`）

1. $O(n\log n+k)$
   先排序 n 个整数，用选择排序，最坏时间复杂度是 $O(n\log n)$
   然后找到中间位置即中位数所在位置，开辟一个 size= $2k$ 的新数组，知道大小最接近的一定在数组 $[\frac{n}{2}-k,\frac{n}{2}+k]$ 中出现，我们调用下面的函数：

```python
def findKNearest(new_data[],k):
    int i=k#smaller pointer
    int j=k+2#larger pointer
    int amt=0
    int[] ans
    while(amt<k):#如果找到的个数不足k个，就继续找
        if(k-new_data[i]<new_data[j]-k):#左侧的更接近
            ans.push(new_data[i])
            i-=1
        else:#右侧的更接近
            ans.push(newdata[j])
            j+=1
```

这个函数使用双指针，最多访问新数组 k 次并进行 k 次比较，所以在 $O(k)$ 内做完
总体在 $O(n\log n+k)$ 内完成
2. $O(n+k\log k)$
在 $O(n)$ 做找中位数算法，找到 `M`
创建一个 new_data 来记录和中位数的大小
`new_data[i]=abs(data[i]-k)`
然后再调用 `select` 算法，找到 `k` 阶元素 `N`，耗费 $O(n)$
以 `N` 做 partial，cost $O(n)$
partial 之后，在比 N 小的部分 `less_data` 做排序，容易知道 `len(less_data)=k` ，做排序最坏情况 cost $O(k\log k)$
最后可以完成。

5.8

//TODO
考虑在多个一维数组或者一个多维数组中进行选择的问题

1. 给定两个有序数组 A, B 和一个整数 k，请设计一个算法用 $O(\log n)$ 的时间找到 $A\cup B$ 中阶为 `k` 的元素
   思路：...有点没思路...

5.10

(加权中位数)

1. 证明如果 `w_i=1/n i=1,2,...n` ，则 `x1,x2,...,xn` 的中位数就是加权中位数
   证明:
   `x1,x2,...,xn` 的中位数 `x_mid`
   假设 `n=2k+1`,则

   $$\sum_{x_i<x_k} w_i = \frac{k}{n} < \frac{1}{2}, \sum_{x_i>x_k} w_i = \frac{k}{n} < \frac{1}{2}$$

   满足加权中位数定义
   假设 `n=2k`，则我们可以选择最靠近中位数的其中一个作为加权中位数，不妨设其阶为 k

   $$\sum_{x_i<x_k} w_i = \frac{k}{n} \le \frac{1}{2}, \sum_{x_i>x_k} w_i = \frac{k-1}{n} < \frac{1}{2}$$

   仍然满足

2. 设计加权中位数算法
   在 $O(n\log n)$ 内做完，可以考虑这样设计:
   先对数组进行归并排序，按照权重从大到小排序，worst time complexity in $O(n\log n)$
   然后，遍历数组，不断加总权重直到 $\sum w_i \ge \frac{1}{2}$，返回 `i`
   则 i 处元素就为加权中位数所在处。因为 i 之前总权重小于 1/2,而加了 i 之后总权重大于或等于 1/2，说明权重比中位数大的元素满足总权重小于等于 1/2，所以找到了加权中位数
   遍历数组只需要在 $O(n)$ 内完成。
   $T(n) = O(n\log n + n) = O(n\log n)$

3. 设计最坏情况时间复杂度为 $\Theta(n)$ 的加权中位数选择算法:
   思路：每次选择一个 pivot 做 partion，然后计算两边权重，再继续更新寻找。

```python
def weighted_median(A, W):
    # A: 数值列表
    # W: 权重列表 (与 A 对应, 且和为 1)
    if len(A) == 1:
        return A[0]
    #找到中位数的中位数做partial
    pivot = median_of_medians(A)

    # 分成三部分
    L, LW = [], []
    E, EW = [], []
    R, RW = [], []

    for a, w in zip(A, W):
        if a < pivot:
            L.append(a)
            LW.append(w)
        elif a > pivot:
            R.append(a)
            RW.append(w)
        else:
            E.append(a)
            EW.append(w)

    wL = sum(LW)
    wE = sum(EW)

    if wL < 0.5 and wL + wE >= 0.5:
        return pivot
    elif wL >= 0.5:
        return weighted_median(L, LW)
    else:
        return weighted_median(R, RW)
```

Chapter 6 (6)

6.2, 6.3, 6.4, 6.5, 6.8, 6.14

6.2

思路：通过基于上一次计算 `m^2` 的结果来优化运算
基于这样一个事实：
假设我们有当前 m 和 m^2。每次我们二分的 `m` 是从前一次加或减了一个 `Δ`：

- 新的 `m' = m + Δ`
- 有公式：

$$(m + \Delta)^2 = m^2 + 2m\Delta + \Delta^2$$

```
所以：
```

$$m\prime^2 = m^2 2 + 2m\Delta + \Delta^2$$

```
如果 `Δ` 是 `2^k`，那么乘法就可以变成移位：

- `2mΔ` = `m << (k+1)`

- `Δ²` = `1 << (2k)`
```

因此，我们只需要保存：

- 当前的 m
- 当前的 m^2

每次更新 m 的时候，只加一些移位量就能得到新的平方，避免重新算一遍。
优化这一点之后，其他的保持不变，$T(n) = 2T\left(\frac{n}{2}\right) + O(1)$，在线性时间内可以完成

6.3

证明红黑树的直接定义和间接定义是等价的

我们直接证明：直接定义的红黑树⇒满足间接定义的红黑树的定义
间接定义的红黑树⇒满足直接定义的红黑树的定义

空树显然符合递归基础定义。
红色节点不能连续出现，即符合：ARB 的定义：根节点为红色（也就是一个红色父节点）之后必然接两个 `RB_{h-1}`。
直接定义中要求黑色深度相同，而在递归定义中，`RB_h` 的两个子树要么是 `RB_h-1` 要么是

`ARB_h`，如果是准红黑树 `ARB_h` 其两个子树都是 `RB_h-1`
归纳可证，黑色深度相同

从间接定义向直接定义考虑是类似的。

所以直接定义和间接定义等价。

6.4 （证明红黑树的平衡性）

引理 1:

1.T 有不少于 `2^h-1` 个内部黑色节点.
用归纳法可证：

- 当 `h = 0`：子树只能是 NIL（不含内部节点）→ 满足。
- 假设对 `h = k` 成立，对 `h = k+1` 的节点，其左右子树黑高至少为 `k`，按归纳有至少 `2^k - 1` 个内部节点 → 加上根节点后为 `2^{k+1} - 1`。

所以一棵黑高为 `h` 的红黑树至少包含 `2^{h} - 1` 个内部节点。

2.T 有不超过 `4^h-1` 个内部节点

借助 3 性质的结论，容易证明：由于普通高度最多是黑色高度的 2 倍，也就是 `h<=2bh`
所以等价于证明T 有不超过 $4^{h/2} - 1$ 个黑色节点。类似归纳可证明。

3.任何黑色节点的普通高度至多是其黑色高度的 2 倍
因为红色节点不相邻，所以红色节点的数目最多和黑色节点一样多。构造一个红黑相间的路径，
则此时考虑黑色 root 节点，此时其普通高度 `h=2bh`，所以普通高度至多是其黑色高度的 2 倍

引理 2:

假设 T 为一颗 `ARB_h` ，则

- T 有不少于 `2^h-2` 个内部黑色节点
- T 有不超过 `(1/2)4^h-1` 个内部节点
- 任何黑色节点的普通高度至多是其黑色高度的二倍。

1. 与引理 1 相似，唯一的区别是 `ARB_h` 不需要加上 root 的黑节点，所以差 1
2. 与引理 1 相似，我们可以通过归纳法证明：

$$M(h) \leq \frac{1}{2} \cdot 4^h - 1$$

基础情况：

- `h = 0`：空树，`M(0) = 0 ≤ 1/2·1 - 1 = -0.5`（成立）
- `h = 1`：一个黑根，`M(1) = 1 ≤ 1/2·4 - 1 = 1`（成立）

归纳假设：

假设对所有小于 h 的情况成立：

$$M(h-1) \leq \frac{1}{2} \cdot 4^{h-1} - 1$$

我们要证明：

$$M(h) \leq \frac{1}{2} \cdot 4^{h} - 1$$

- 构造左右子树黑高度为 `h - 1`，即都是 `RB_h-1`
- 所以有两个子树，根据 1 中结论每个最多有 $4^{h-1}-1$ 个内部节点。
  总的内部节点个数为：

  $$2(4^{h-1}-1)+1(root) = \frac{1}{2}4^{h}-1$$

  證畢
3. 任何黑色节点的普通高度至多是其黑色高度的 2 倍
   证明构造和引理 1 完全类似。略去。

6.5

给定一个有 `n` 个互不相同的有序整数 `[a1,a2,...,an]` 的序列，请设计算法判断是否存在某个下标 i 满足 `ai=i`，

思路：选取中位数作为 pivot，由于有序，自动 partial 为两部分。检查：

- `pivot<index(pivot)` ，说明 pivot 左侧都没有，从右侧找
- `else`：从左侧找。
  整理算法：

```python
def findIndexValueEqual(data[],left,right):
    int mid_idx=(left+right)/2
    if(left>right):
        return NOTFOUND
    int mid=data[(left+right)/2]
    if(mid==mid_idx
        return mid;

    if(mid<len/2):
        return findIndexValueEqual(data,mid_idx+1,right )
    else:
        return findIndexValueEqual(data,left,mid_idx-1)
```

该算法平均复杂度在 $O(n)$ 级别

6.8

请对下列问题找到合适的算法。可以使用已有的排序算法

1. S 是由 n 个整数组成的数组，并且未排序。请给出算法用来找到整数对 $x, y \in S$ 使得 $|x-y|$ 最大，最坏情况下应该为 $O(n)$

调用已有选择算法，分别找到最大值和最小值，总共花费 $O(n)$
这样就找到了 $|x-y|$ 最大值

2. S 是由 n 个整数组成的有序数组，请给出算法完成相同的任务，最坏复杂度为 $O(1)$

直接返回 `data[0]` 和 `data[len-1]`，即最前和最后元素

3. S 是 n 个整数组成的数组，未排序，给出算法找出整数对 $x, y \in S$ 使得 $|x-y|, x \neq y$ 最小，最坏情况 $O(n \log n)$

采取分治的方法设计。为了使划分平衡，先用 $O(n)$ 时间找到中位数 mid 做 partial，然后对于大于 mid 和小于 mid 的两部分 L (less)和 M(more)，分别计算其中每个元素和 mid 的绝对值大小，记其中最小的为 min2mid
然后递归的在 less 和 more 中找最小，分别记为 `minInL, minInM`
divide 以上做，如何 conquer(合并)：`return min(min2mid,minInL,minInM)即可

4. S 是由 n 个整数组成的有序数组给出算法找出整数对 $x, y \in S$ 使得 $|x-y|, x \neq y$ 最小，最坏情况 $O(n)$
   容易证明，对于有序数组，要寻找的整数对一定出现在相邻整数中。

```python
def findMinDis(data[]):
    int tmp[]=[abs(data[i+1]-data[i]) for i in range(len(data)-1)]
    return findMinIndex(tmp)
```

所以只需要计算 $|x-y|$ 最小值即可，创建新数组和找最小都可以在 $O(n)$ 内完成，即得到答案。

6.14（数组上的局部最小元素）

1.证明
假设不存在局部最小元素，这意味着 $\forall i \in [1, n] \wedge i \in \mathbb{Z}$，它至少大于它的一个邻居。
数组中的元素至少大于一个邻居有以下选择，考虑相邻的三个元素 a,b,c

1. a, c 都大于 b，矛盾，不可能。
2. `left neighbor of a < a < b < c` 此时可能.
3. `a > b > c> right neighbor of c` 可能
4. `a<b>c` 可能
   所以数组可能是单峰的（中间存在最大值），或者单调的（递增，递减），共三种情形。

- 单峰，则有 `A[2]<A[3]`，题目条件有 `A[1]>=A[2]`，矛盾
- 单调递增，则有 `A[2]<A[3]`，题目条件有 `A[1]>=A[2]`，矛盾
- 单调递减，则有 `A[n-2]<=A[n-1]`，题目条件有 `A[n-1]<=A{n}`，矛盾
  故假设错误，一定存在局部最小元素。

Chapter 15 （2）

- 15.1, 15.3

15.1

假设一个并查集中有 n 个元素，并查集指令序列的长度为 l，请对于并查集的不同实现方法给出具体的算法实现并分析代价

1. 基于矩阵：$O(nl)$
2. 基于数组：$O(nl)$

- 基于矩阵
  矩阵：构造一个 $n \times n$ 的矩阵，若元素 a，b 属于一个等价类，则 $(a, b) = 1$，否则 $(a, b) = 0$

`union(a,b)`：

```python
def union(a,b,data[][]):
    data[a][b]=1
    #更新关系
    for i in len(data[a][]):
        if(data[a][i]==1):
            data[b][i]=1
    for j in data[][a]:
        if(data[j][a]==1):
            data[j][b]=1
```

`find(a):
矩阵实现没有维护代表元，我们需要遍历所有和 a 有关系的元素

```python
def find(a):
    equivs[]
    for i in len(data[a][]):
        if(data[a][i]==a):
            equivs.append(elements[i])
    return equivs
```

这种实现下，`union` 的代价是遍历两次 n 的数组，代价为 $O(n)$ `find` 的代价也是遍历数组。

故代价为 $O(nl)$

- 基于数组
  数组 $E[1\dots n]$ 的每个位置 $E[i]$ 存储的是元素 a_i 所在等价类的代表元.
  `union(a,b)`

```python
def union(a,b,E[]):#将b的代表元挂为a
    #Assume a, b is index
     for i in range(len(E)):
        if(E[i]==E[b]):#和b有相同代表元
            E[i]=E[a]#将所有b的等价类代表元改为a
```

`find(a)`
`return E[a]`
在这种实现下，由于 `find` 在 $O(1)$ 内完成，`union` 需要遍历一次数组，代价为 $O(n)$，总体代价为 $O(nl)$

15.3

维护一个并查集来实现这个检查。
`union-find` 的 ADT 应该包括 `find` 和 `union` 两个操作。

`等于` 是一种等价关系（自反，传递，对称），但是 `不等于` 不是，

PYTHON

```python
def equConstraintCheck(constraint[]):
    #constraint存储若干约束字符串

    notEquivOf[][]
    for i in constraint[]:
        if i is "a==b": #仅做一个示例
            if(a in notEquivOf[b]):
                return FALSE
            if(b in notEquivOf[a]):
                return FALSE
            #不冲突可以添加
            union(a,b)
        if i is "a!=b":
            EofA=find(a)
            EofB=find(b)
            if(EofA==EofB):
                return FALSE
            else:
                notEquivOf[a].append(b)
                notEquivOf[b].append(a)
    return TRUE
```

Chapter 16 (1)

16.1

有一个大小为 `n` 的哈希表，close address, n 个 key，hash 到某个 address 概率相同。证明正好有 k 个关键字 hash 到某个地址的概率是

$$Q_k = \left(\frac{1}{n}\right)^k\left(1-\frac{1}{n}\right)^{n-k}\binom{n}{k}$$

- proof
  这是一个伯努利模型. 所有 key hash 到某个特定 address 的概率 $p=\frac{1}{n}$，相当于进行 k 次重复伯努利实验，服从伯努利分布，由此立得

Chapter 18（2）

18.1

1. 证明算法正确性
   找到不变式：
   栈中的每一个元素，都比其更靠近栈顶的元素高度更大
   归纳证明：
   BASE
   只有一个元素的时候显然符合
   I.H
   假设栈中元素有 `n-1` 个时候符合
   I.STEP
   证明栈中有 `n` 个元素符合
   proof:
   for all `E` in `S[0:n-1]`, `height of E` 都大于它右边的元素，得到大于栈顶元素
   由算法 `while(S not empty and A[i]>S.TOP()) do` 可知，算法插入新元素时，`new_E` 一定小于栈顶元素，从而也满足栈中的每一个元素，都比其更靠近栈顶的元素高度更大
   得证。
   证明了这个性质：由于栈的次序严格满足楼的东西次序，而高度又满足，所以栈中的元素都是“LAKE-VIEW 楼”
   容易知道该算法不会漏掉 LAKE-VIEW 楼，即它只弹出不是 LAKE-VIEW 楼的元素（比右边楼高度小），此外没有别的删除操作。
   得证。
2. 平摊分析
   本算法主要进行两项操作：压栈和出栈。所有出栈过的元素（被淘汰，不是湖景房）的不会再被压栈。
   对于 `A` 中某元素，其要进栈(记为操作 APUSH，和栈 PUSH 区分)，算法会做大小检查，可能最多弹出栈中所有的元素。
   我们对 PUSH收取 1 account cost，也就是“预收”了弹出栈的 cost

- `APUSH`
  - actual cost: 1(push)+several 0(POP total cost)
  - accout cost:-several 0(POP total cost)
  - total cost:1
- `POP`
  - actual cost: 1
  - account cost: -1(PUSH 预收)
  - total cost: 0

- `PUSH`
  - actual cost: 1
  - account cost: 1
  - total cost: 2
  由于进栈的元素个数大于等于被弹出的元素个数，所以总 account cost
  $$\text{total account cost} = \text{amt of push} * 1 + \text{amt of pop} * (-1) \geq 0$$
  会计成本非负，平摊分析有效。算法在 $O(n)$ 内完成

18.3

考虑二进制的表示方法，容易思考这个问题
数组：第 i 个数组有 2^i 个元素，要么满要么空，可以构建这样一个二进制串来表示：
如：有 5 个元素，5 的二进制表示为 `101`，二进制串最低位对应第 0 个数组。对应位为 1 为满，对应位为 0 为空。
所以：算法描述的查看是否为空，本质就是+1 的进位过程。
1011→1100 产生新元素 cost 1，在第一个 0 位停下，cost $1 + 2 + 2^2$
分析：元素只会向下一个数组移动，所以我们可以在其创建的时候预收取"费用"

- 创建新数组
  创建的新数组里面只有一个元素，其最多被合并 $\log n$（n 是元素个数）次
  - actual cost= 1
  - account cost = `log n`
- 合并：
  - actual cost = 2m
  - account cost = -2m
  - total cost=0
  以上平摊是可行的，这是因为合并数组的代价 $2m$ 实际上是两个数组遍历发生的，而我们在创建数组时候，已经为每一个元素分配了其对应合并次数最多的成本，能够完全 cover 。

而一次插入，其进行创建新数组和若干次合并，若干次合并的代价已经在之前创建数组中其他元素中收过了，且会计成本和非负（我们高估了合并次数，给了足够多的会计成本）
所以时间复杂度为 $O(\log n)$

Chatper 19 (3)

19.1, 19.3, 19.4，19.5, 19.6，19.10

19.1

使用决策树证明选择问题（选择任意第 k 大的元素）的最坏情况时间复杂度的下界是 $O(\log n)$

证明：要选择第 k 大的元素，必然要知道这个元素和其他所有元素的大小关系，这至少是一个 $n-1$ 的答案空间。
决策树进行一次比较，产生两个分支，最后得到 2^h 个叶子节点，每个叶节点都是一种答案。所以，要得到第 k 大元素，决策树必须一定有"答案"，即叶子结点个数必须大于答案空间，应该有

$$2^h > n-1$$

即

$$h > O(\log n)$$

所以答案下界是 $O(\log n)$

19.3

已知数组 $A[1...n]$ 中至多有 1 个逆序对，现在需要将数组中的元素排序，请用对手论证证明：任何算法在最坏情况下至少需要 $n-1$ 次比较，才能完成数组中的元素的排序。
proof：
用反证法。假设有一个算法，可以在 $n-2$ 次比较完成排序，我们从对手角度，让数组有 1 个逆序对（逆序对数量最大化），算法能做的最好表现，就是让 $n-2$ 次比较全部是逆序对中的一个元素，和其他元素的比较，其必然有一个元素没有比较到。我们从对手的角度，令这个元素就是逆序对的另一个元素，则算法中仍存在逆序对，排序没有完成。

19.4

对于 7.4 的芯片检测问题，用对手论证证明：如果坏芯片的数目不少于总数的一半，则任何算法都不能确保正确检测所有芯片的好坏
芯片比较的规则如下

A 芯片报告	B 芯片报告	结论
B 是好的	A 是好的	都是好的，或者都是坏的
B 是好的	A 是坏的	至少一片是坏的
B 是坏的	A 是好的	至少一片是坏的
B 是坏的	A 是坏的	至少一片是坏的

在 7.4构造芯片算法中，我们试图维护一个重要的不变式：好芯片至少比坏芯片多一片，在这种情形下，该不变式不成立，所以不能确保正确检测所有芯片的好坏。
用对手论证来具体说明这一点：

对于任何算法而言，确保能正确检测所有芯片的好坏，就意味着找到一张一定是好芯片的芯片，然后用它去检查。
对于后三种情况，算法只能确定删掉 B 和 A 之后，好芯片个数-坏芯片个数增加（因为至少一片

是坏的，也可能两个都是坏的）
作为对手，我们可以构造这样一个情况，即总让算法删除的是只有一个坏芯片。所以当后三种情况全部删除后，由条件 **坏芯片的数目不少于总数的一半**，我们的坏芯片个数仍然大于好芯片
下面只剩下比较结果是第一种的芯片们，算法试图继续删除找到一张好芯片，但是这是无法做到的：作为对手，我们可以每次让他删除的都是好芯片，则无论剩下的牌是奇数还是偶数张，算法最后都无法找到好芯片。且他也无法确定自己拿到的是好芯片还是坏芯片。