Assignment1

T1.

You are interested in analyzing some hard-to-obtain data from two separate databases. Each database contains n numerical values, so there are $2n$ values total and you may assume that no two values are the same. You'd like to determine the median of this set of $2n$ values, which we will define here to be the n^{th} smallest value.

However, the only way you can access these values is through *queries* to the databases. In a single query, you can specify a value k to one of the two databases, and the chosen database will return the k^{th} smallest value that it contains. Since queries are expensive, you would like to compute the median using as few queries as possible.

Give an algorithm that finds the median value using at most $O(\log n)$ queries.

Solution1：

1. natural language :

经分析，想找到两个数组中的（共 2n 个数字）中位数，其最终的结果形式必然是：

$$n_{\text{中位数}} = \frac{n_1 + n_2}{2}$$

其中 n_1，n_2 一定是将 2 个数据库按序排列后相邻的两个数，故找到其中一个数即可。

[1] 设定第一个数据库中的最小的数下标为 min = 1，最大的数字为 max =n

[2] 在第一个数据库中找到第 a（a=（min + max）/2）小的数字 Ka，则比 Ka 小的数有 a-1 个，比 Ka 大的数有 n-a 个；

[3] 在第二个数据库中找到第 b（b= n-a+1）小的数字 Kb，则比 Kb 小的数字有 n-a 个，比 Kb 大的数字有 a-1 个；

[4] 若 K_{b-1}<=Ka 且 K_{b+1}>=Ka，则中位数的其中一个必为 K_a(暂且看做 n_1)，计算退出。

[5] 若 K_{b-1}>Ka，则 min = a，继续从[2]开始执行；

[6] 若 K_{b+1}<Ka，则 max = a，继续从[2]开始执行；

[7] 比较 K_b 与 K_{a-1} 和 K_{a+1} 之间的关系，从而确定另外一个 n2。

2. pseudo-code :

先将如下定义：

database1[k] ：第一个数据库第 k 小的数字

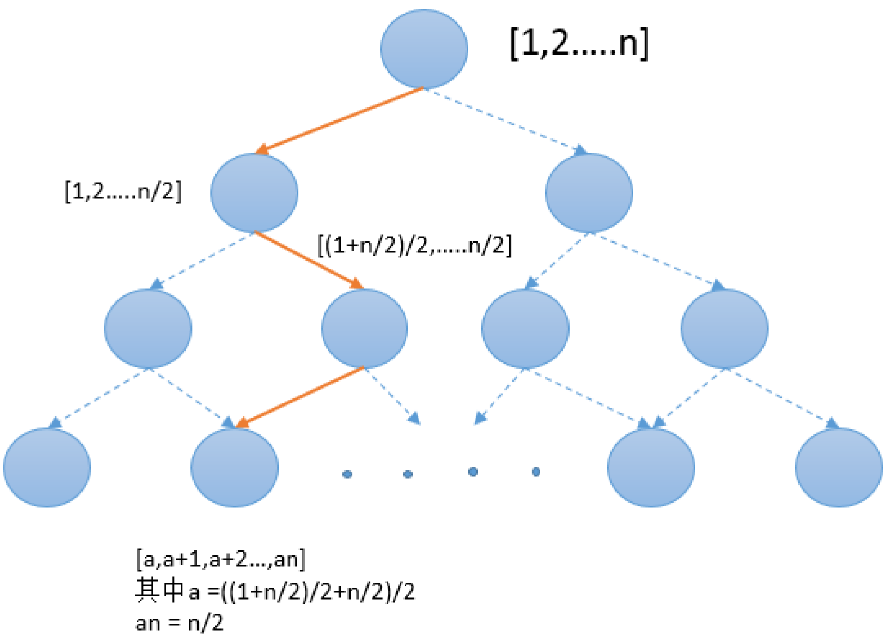
规定 database[0] = 无穷小 database[n+1] = 无穷大

```
min = 1
max = n
while（true）
    if n%2 == 1
```

```
        a = (min + max)/2 + 1
    else
        a = (min + max)/2
    b = n - a + 1
    if database2[b-1]<=database1[a] && database2[b+1]>=database1[a]
        return a
        break
    else
        if database2[b-1]>database1[a]
            min = a
        else
            max = a

if database1[a-1] <= database2[b] <= database1[a+1]
    res = (database2[b] + database1[a])/2
else if database1[a-1] > database2[b]
    res = (database1[a-1] + database1[a])/2
else database2[b] > database1[a+1]
    res = (database1[a+1] + database1[a])/2
```

3. subproblem reduction graph :



4. Prove the correctness :

在第一个数据库中找到第 a 小的数字 Ka，则比 Ka 小的数有 a-1 个，比 Ka 大的数有 n-a 个；在第二个数据库中找到第 b（b= n-a+1）小的数字 K_b，则比 K_b 小的数字有 n-a 个，比 K_b 大的数字有 a-1 个。保证若 K_{b-1}<=Ka 且 K_{b+1}>=Ka

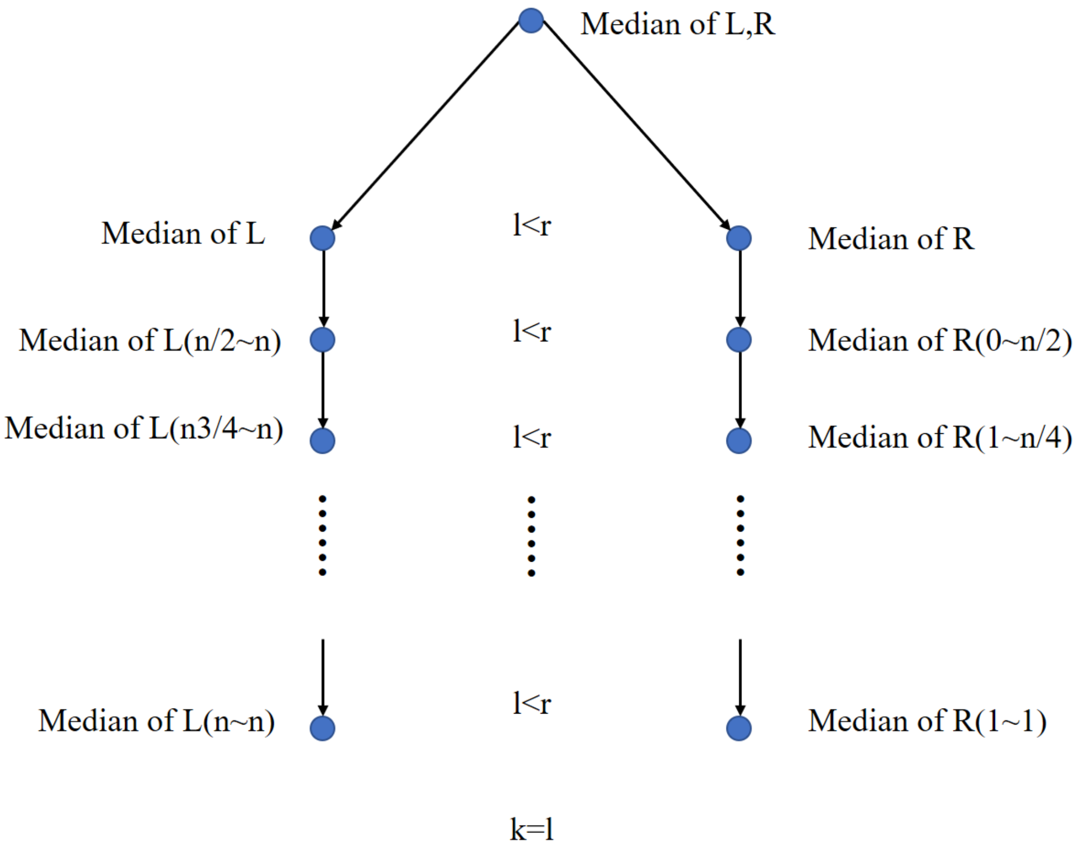
则在两个数据库中有：
比 Ka 小的数至少有 n-1 个，比 Ka 大的数至少有 n-1 个
此时,已经找到 n_1,n_2 其中之一，接下里只需要比较 K_b 与 Ka_{-1} 和 K_{a+1} 之间的关系，找出 n_2 即可：
若 database1[a-1] <= database2[b] <= database1[a+1]
则说明同时比 Ka 和 K_b 小的数有 n-1 个，同时比 Ka 和 K_b 大的数有 n-1 个。
则中位数是：$(Ka + K_b)/2$
若 database1[a-1] > database2[b]
则说明同时比 Ka 和 K_{a-1} 小的数有 n-1 个，同时比 Ka 和 K_b 大的数有 n-1 个。
则中位数是：$(Ka + K_{a-1})/2$
若 database1[a+1] < database2[b]
则说明同时比 Ka 和 K_{a+1} 小的数有 n-1 个，同时比 Ka 和 K_b 大的数有 n-1 个。
则中位数是：$(Ka + K_{a+1})/2$
5. Analyse the complexity:
因为循环体采用二分计算方法，且循环体内部是常熟数量级的运算，故该算法的时间复杂度为二分算法的时间复杂度 O（logn)。
Solution2：
算法描述：（[]在此为取整符号）
记要求的目标中位数为 k，两个大小为 n 的数组分别记为 L 和 R，首先查询 L 中第[n/2]小的数记为 l1，然后查询 R 中第[n/2]小的数记为 r1，比较 l1 和 r1 的大小，不妨设 l1<r1，则说明 L 中不大于 l1 的[n/2]个数都比要求的 k 小，而 R 中大于 r1 的 n-[n/2]个数都比要求的 k 大。所以两次查询便将搜索范围由 2n 缩小至 n。第二次便查询 L 中第[3n/4]小的数记为 l2,R 中第[n/4]小的数记为 r2，比较 l2 与 r2 的大小，若 l2<r2，说明 L 中不大于 l2 的[3n/4]个数都比 k 小，R 中大于 r2 的 n-[n/4]个数大于 k，此时搜索范围由 n 变为[n/2]。然后比较 l1 和 l2,r1 和 r2。如果不全相等，则将 l2 赋值给 l1，r2 赋值给 r1，说明搜索空间还不够小，如果相等，则说明已搜索完全，那么 k 就等于 l1。以此类推，每次都可以缩小一半的搜索空间，直到搜索结束。
伪代码：

```
Input:L,R
(1) l1=Query(L,[n/2]);
(2) r1=Query(L,[n/2]);
(3) i=2;nl=1/2;nr=1/2;
(4) while(True){
        if(l1<r1){
            l2=Query(L,[n*(nl+(1/2)^i)]);
            r2=Query(R,[n*(nl-(1/2)^i)]);
        }
        else{
            l2=Query(L,[n*(nl-(1/2)^i)]);
            r2=Query(R,[n*(nl+(1/2)^i)]);
        }
        i++;
        if(l1==l2){
```

```
            if(r1==r2){
                if(l1<r1){
                    k=l1
                }else{
                    k=r1;
                }
                break;
            }
        }
        else{
            l1=l2; r1=r2;
        }
    }
    return k;
Output:k
```

子问题分解图：

本图画的是所有 l 都小于 r 的情况。



算法证明：

当搜索范围为 2 的时候，即 L 中一个元素，R 中一个元素，此时，K=(l+r)/2, 成立。

若当搜索范围为 n 时，算法成立，那么当搜索范围为 2n 时：L 中有 n 个元素，R 中有 n 个元素，根据算法可将 L，R 各分为两个部分，并根据 L,R 中位数

大小关系各去掉一半的搜索空间，则变成了搜索范围为 n 时的问题，所以 2n 时也成立。综上所述，该算法能够得到两个大小为 n 的数据库的中位数。

复杂度分析：

由于 T(2n)=T(n)+T(n),若每次复杂度为 n，则整体复杂度为 O(nlogn),而本题中，每次分解都只有 2 次查询操作，所以整体复杂度为 O(2log2n)。

T2：

Find the k^{th} largest element in an unsorted array. Note that it is the kth largest element in the sorted order, not the k^{th} distinct element.

INPUT: An unsorted array A and k.

OUTPUT: The k^{th} largest element in the unsorted array A.

Solution1:

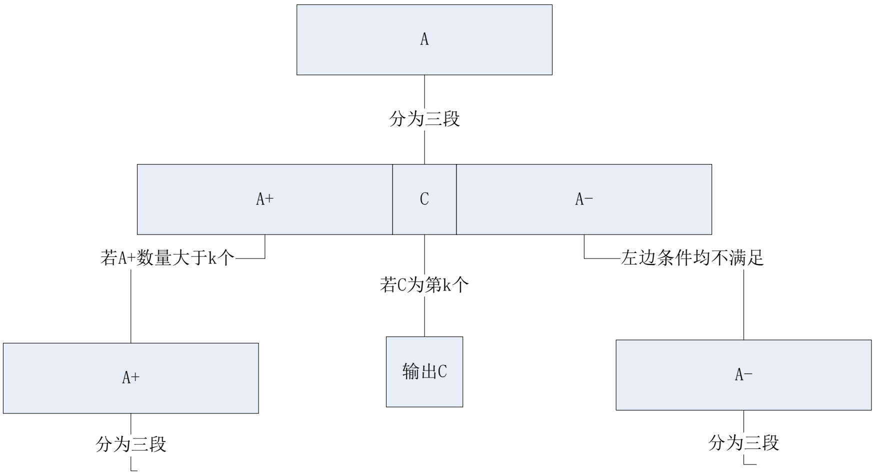
Problem-solving ideas

从 A 中选择一个数 S，将比 S 大的数放在 S 前，将比 S 小的数放在 S 后，若 S 之前有 k-1 个，则 S 就是所求结果。否则若 S 之前有大于 k 个数，则在 S 之前找，若 S 之前有小于 k 个数，则在 S 之后找。

Pseudo-code

```
public int top-kth(List A, int k){
    int sap = A(A.length/2);//从 A 中随意选一个数
    List front = new List();//大于 sap 队列
    List end = new List();//小于 sap 队列
    int time = 0;//等于 sap 队列
    //将 A 分成三队，大于 sap 的，小于 sap 的，等于 sap 的
    for(int i=0; i<A.length; i++){
        if(A(i)>sap){
            front.add(A(i));
        }else if(A(i)<sap){
            end.add(A(i));
        }else{
            time++;
        }
    }
    if(front.length>k){
        //从大于 sap 队列中找
        return top-kth(front,k);
    }else if(front.length+time>k){
        //输出
        return sap;
    }else{
        //从小于 sap 队列中找
        return top-kth(end,k-front.length-time);
    }
}
```

Subproblem reduction graph



Prove the correctness

根据快速排序的算法，每次确定一个元素的位置，在该元素左侧均大于该元素值，在该元素右侧均小于该元素值。通过比较，我们能确定第 k 大的值在 A 中的哪个范围，随着递归深入，一定能找到第 k 大的值。

Analyse the complexity

时间复杂度是 O(cn)

Sulotion2：

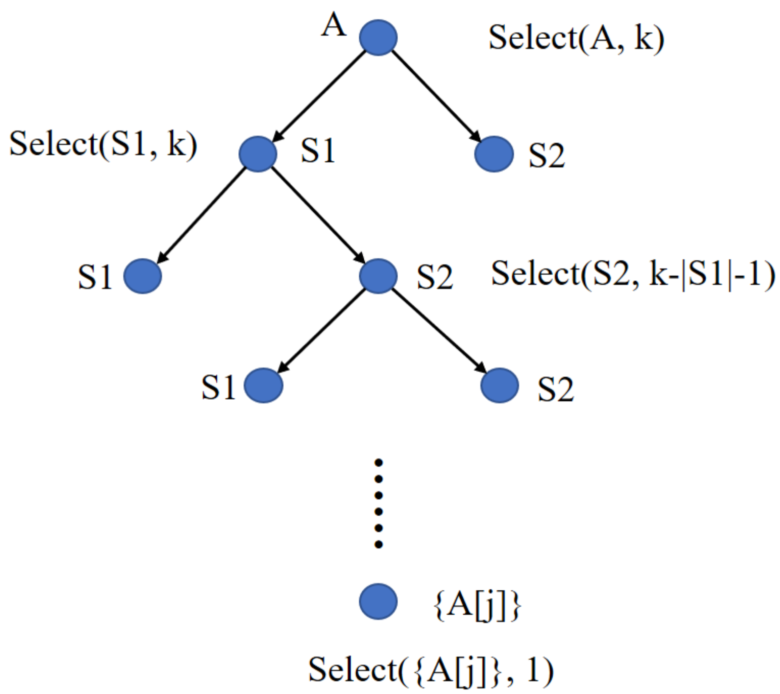
算法描述：

首先，我们需要选择一个指针 A[i]，将数组中所有其他的数与 A[i]进行比较，将大于等于 A[i]的数放在 S1 中，其余的放在 S2 中，求出 S1 的大小，与 k 比较，如果大于 k，则进行在 S1 中选择第 k 大的数，如果小于 k，则进行在 S2 中选择第(k-|S1|-1)大的数。

伪代码：

```
Select (A, k)
(1) Choose an element A[i] from A as a pivot; use ‘a’ to note it;
(2) S1={};S2={};
(3) for j=1 to n do{
        if (j==i){continue;}
        if (A[j]>=a){ S1=S1U{A[j]};}
        else{ S2=S2U{A[j]};}
    }
(4) if (|S1|==k-1){ return k}
(5) if (|S1|<k-1){return Select(S2,k-(|S1|+1));}
(6) if (|S1)>k-1){return Select(S1,k);}
```

子问题分解图：



算法证明：

当数组中只有一个元素，选择第 1 大的元素时，Select({A[j]},1)=A[j]，正确。

若当数组中有|S2|个元素时，选择第 k-|S1|-1 大的数时，算法正确，则当数组中含有|S1|+|S2|+1 个元素，选取其中第 k 大的数时，根据算法，S1 中所有的数都比 a(选取的指针元素)大，S2 中所有的元素都比 a 小，且|S1|<k-1,那么要选取的第 k 大的元素必然在 S2 中，并且是 S2 中第(k-|S1|-1)大的元素，即 Select(A,k)=Select(S2, k-|S1|-1)，由于后者是正确的，所以 Select(A,k)也是正确的。

综上所述，算法是正确的。

复杂度分析：

最坏情况，每次都选到最大的作为指针，复杂度为：

$T(n)=T(n-1)+O(n);T(n)=O(n^2);$

最好情况，每次都选到中位数作为指针，复杂度为：

$T(n)=T(n/2)+O(n);T(n)=O(n);$

如果能选到与中位数比较接近的数作为指针：

$|S1|>\varepsilon n$, $|S2|>\varepsilon n$,其中，$\varepsilon>0$;

$T(n)\leqq T((1-\varepsilon)n)+O(n)\leqq cn+c(1-\varepsilon)n+ c(1-\varepsilon)^2n+\cdots=O(n)$.

T3：

Consider an n-node complete binary tree T, where $n = 2^d - 1$ for some d. Each node v of T is labeled with a real number x_v. You may assume that the real numbers labeling the nodes are all distinct. A node v of T is a *local minimum* if the label x_v is less than the label x_w for all nodes w that are joined to v by an edge.

You are given such a complete binary tree T, but the labeling is only specified in the following *implicit* way: for each node v, you can determine the value x_v by *probing* the node v. Show how to find a local minimum of T using only $O(\log n)$ *probes* to the nodes of T.

Solution1：

从树根开始，若比两个儿子都小，则树根就是所求的局部最小值，否则对较小的儿子重复该操作（此时根是比这个儿子大的）。

当执行到某节点的时候，如果该点没有儿子，则只有一个比它大的父亲，那么他就是局部最小值，如果有儿子，两个都比它小，那他也是最小值，否则将较小的儿子作为下一个考虑的节点，继续重复该操作，总会遇到叶子终止或者是遇到两个儿子都比他小的情况从而终止。

ALGORITHM 3: *FindLocalMinimum*

$Node \leftarrow Root\_of\_Tree$

while $Node\_Has\_Son$ **do**
  if $Value\_of\_Left\_Son > Value\_of\_Node$ **and** $Value\_of\_Right\_Son > Value\_of\_Node$ **then**
    return $Node$
  end
  if $Value\_of\_Left\_Son > Value\_of\_Right\_Son$ **then**
    $Node \leftarrow Right\_Son$
  end
  else
    $Node \leftarrow Left\_Son$
  end
end
return $Node$

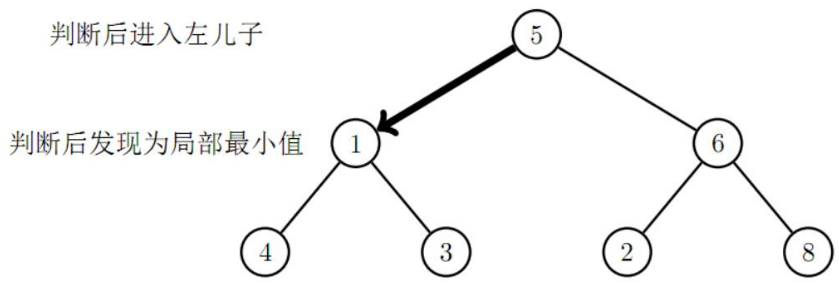


图 3: subproblem reduction graph of 3 *Divide and Conquer*

最差的情况下是一路走到叶子节点，由于是完全二叉树，最差情况就是树深 log(n)。

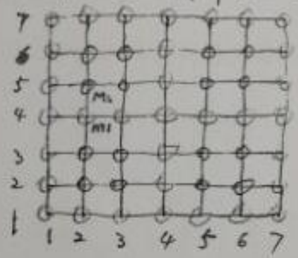
T4：

Suppose now that you're given an $n \times n$ grid graph G. (An $n \times n$ grid graph is just the adjacency graph of an $n \times n$ chessboard. To be completely precise, it is a graph whose node set is the set of all ordered pairs of natural numbers (i, j), where $1 \leqslant i \leqslant n$ and $1 \leqslant j \leqslant n$; the nodes (i, j) and (k, l) are joined by an edge if and only if $|i-k|+|j-l|=1$.)

We use some of the terminology of problem 3. Again, each node v is labeled by a real number x_v; you may assume that all these labels are distinct. Show how to find a local minimum of G using only $O(n)$ probes to the nodes of G. (Note that G has n^2 nodes.)

Solution1：

3. 找一个 $n\times n$ 的格子图的局部最小值.



算法描述：考查 $x=1$, $x=7$, $x=4$, $y=1$, $y=7$, $y=4$ 这6条线上的交点，并找出最小值的点 m.

这6条线将区域划分为 A_1, A_2, A_3, A_4 四个小区域.

若 m 比周围4个点都小，则 m 就局部最小值.

若 m 周围有一个点 n, n 的值比 m 小，则在 n 所有的小区域中找.

伪代码：

```
FindLocalMin(G){
    m=找到6条线的交点中的最小点;
    if( m < 周围的点){
        return m;
    }else if(m > 大于周围点n){
        FindLocalMin(n点所在的图Gi)
    }
}
```

正确性：一个网格图中一定存在局部最小值。在所有红色点中，若最小值为 m_1，若 m_1 比周围四个点都小，则 m_1 必定是局部最小值。若 m_1 比 m_2 大，则 m_2 一定比左上部分一网格的点都小，则左上网格一定存在局部最小值.

复杂度：

$$T(n) = T(\frac{n}{4}) + c$$

$\therefore O(n)$.

T5：

Given a convex polygon with n vertices, we can divide it into several separated pieces, such that every piece is a triangle. When $n = 4$, there are two different ways to divide the polygon; When $n = 5$, there are five different ways.

Give an algorithm that decides how many ways we can divide a convex polygon with n vertices into triangles.

Solution1：

1. natural language :

把一个凸 N 边形的各个顶点按照顺时针分别编上 1，2，……，N。

顶点 1，顶点 N 和顶点 I（I∈[2, N-1]）能够构成一个三角形 S。

这样凸 N 边形就被分成三部分：一个三角形 S、一个 I 边形和一个 N+1-I 边形（I, N+1-I∈[2, N-1]）。

因此，凸 N 边形分为三角形总数 Total(N)等于 I 边形的分法总数乘以 N+1-I 边形的分法总数之积，还要在 I 分别取 2，3，……，N—1 时都累加起来。

2. pseudo-code :

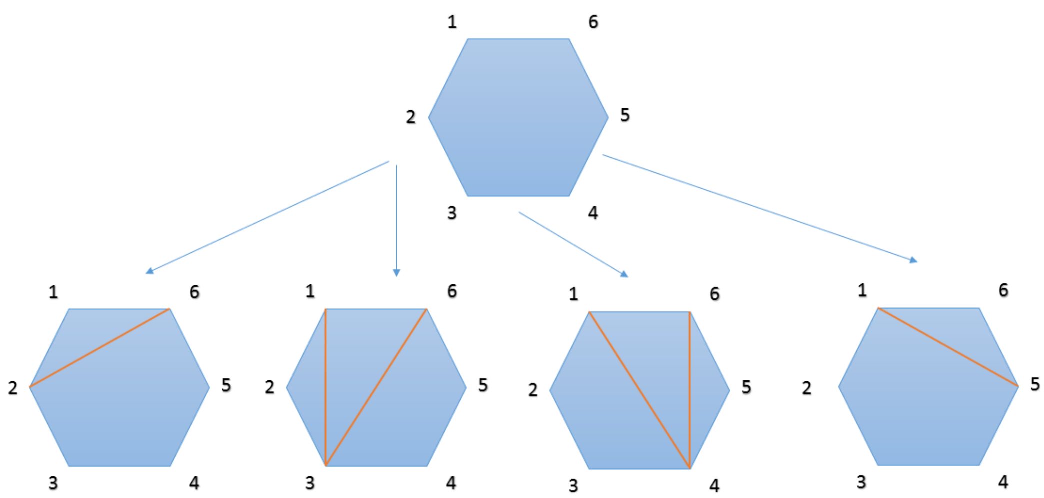
易得，当顶点 I 选取为 2 和 N-1 时，凸边形被分成一个三边形和一个 N-1 边形，即 f(2)表示当凸边形边数为 2 时的情况，我们规定为 f（2） = 1,且 f（3）=1

```
int Sum(int n){
    if n == 2 || n == 3
        return 1

    int count = 0;
    for (int i = 2; i < n; i++)
         count += Sum(i) * Sum (n + 1 -i);

    return count;
}
```

3. subproblem reduction graph :



4. Prove the correctness :

递推公式如下：

Total(N) = sum{ Total(I)*Total(N+1-I) | for I=2 to N-1}   if N>=4

Total(2) = Total(3) = 1
当 2 点的多边形视为蜕化的多边形，定义其 Total(2)=1，是为递推公式推导用。但按题目意思当 N=2 时输出无解。

5. Analyse the complexity:

若选取 **pivotValue** 为中位数，Partition 的线性期望时间 O(N). 递归中时间复杂度为 T（n） = T(2)T（N+1-2）+ T(3)T（N+1-3）+…+ T(n-1)T（2）。如果事先用数组存储 T（2），T（3），….T（n-1）。则计算 T（N）的总时间复杂度为 O(N)，否则为 O（N^3）.

T6：

Recall the problem of finding the number of inversions. As in the course, we are given a sequence of n numbers $a_1, \cdots, a_n$, which we assume are all distinct, and we difine an inversion to be a pair $i < j$ such that $a_i > a_j$.

We motivated the problem of counting inversions as a good measure of how different two orderings are. However, one might feel that this measure is too sensitive. Let's call a pair a *significant inversion* if $i < j$ and $a_i > 3a_j$. Given an $O(n \log n)$ algorithm to count the number of significant inversions between two orderings.

Solution1：

1) 算法描述：

根据题目描述我们可以依照归并求逆序数的方法来求解 i < j 且 a i > 3a j 这种扩展的逆序数，我们将初始数组 A 分成两份 L 表示 A[0….n/2] ，R 表示 A[n/2+1……n-1]，序列的逆序数用 N 来表示，那么 N=N(A1)+N(A2)+M(A1A2).

首先我们设置 3 个索引分别为 I,j,k.i 索引 L,j 和 k 索引 R,在归并的过程中我们比较 L[i]和 R[j]：

1.如果 L[i]<R[j],然后我们再判断 L[i]>3R[k]是否成立，若成立，N+=len(L)-i,k=k+1。若不成立，我们就把 L[i]放到合并时的排序数组 A 中，i=i+1。

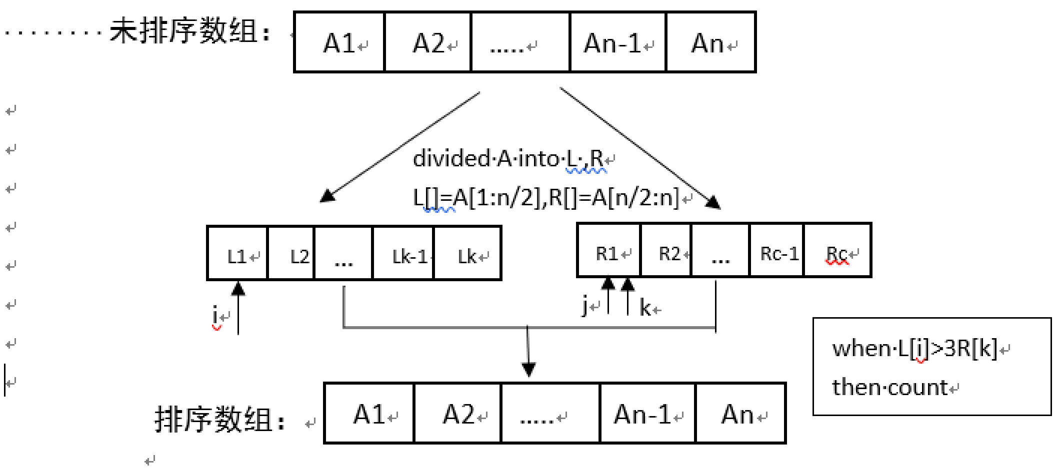
2.如果 L[i]>R[j]，我们就将 R[j]放在合并时排序数组 A 中，j=j+1。

2) 伪代码：

```
def InversionNum(A):
if len(A)<=1
return (0,A)
else:
divide A into L and R
N1,L= InversionNum(L)
N2,R= InversionNum(R)
N3,A=Merge(L,R)
return (N1+N2+N3,A)
def Merge(L,R):
    A=[]
N=i=j=k=0
for index L[i] and index R[j] and R[k]:
    if L[i]<R[j]:
        if L[i]>3*R[k]:
```

```
                N+=len(L)-i
                  k=k+1
            else:
                  A.append(L[i])
                    i=i+1
        else:
              A.append(R[j])
              j=j+1
    return(N,A)
```

3) 子问题分割图：



4) 正确性验证：

根据一般的求逆序数的方法，扩展的逆序数求法只是在计算逆序数对时的计数方法不一样，根据算法描述只有当 L[i]>3R[k]时我们才计数，所以该方法也是正确的。

5) 时间复杂度分析

算法的复杂度：T(n)≤2T(n/2)+cn,其中 cn 为 Merge(A1,A2)的复杂度，因为每一次递归，数量就减少一半，所以最终的到的复杂度为 O(nlogn).

T7.

A group of n ghostbusters is battling n ghosts. Each ghostbuster is armed with a proton pack, which shoots a stream at a ghost, eradicating it. A stream goes in a straight line and terminates when it hits the ghost. The ghostbusters decide upon the following strategy. They will pair off with the ghosts, forming n ghostbuster-ghost pairs, and then simultaneously each ghostbuster will shoot a stream at his chosen ghost. As we all know, it is very dangerous to let streams cross, and so the ghostbusters must choose pairings for which no streams will cross. Assume that the position of each ghostbuster and each ghost is a fixed point in the plane and that no three positions are collinear.

1. Show that there exists a line passing through one ghostbuster and one ghost such the number of ghostbusters on one side of the line equals the number of ghosts on the same side. Describe how to find such a line in $O(n \log n)$ time.

2. Give an $O(n^2 \log n)$-time algorithm to pair ghostbusters with ghosts in such a way that no streams cross.

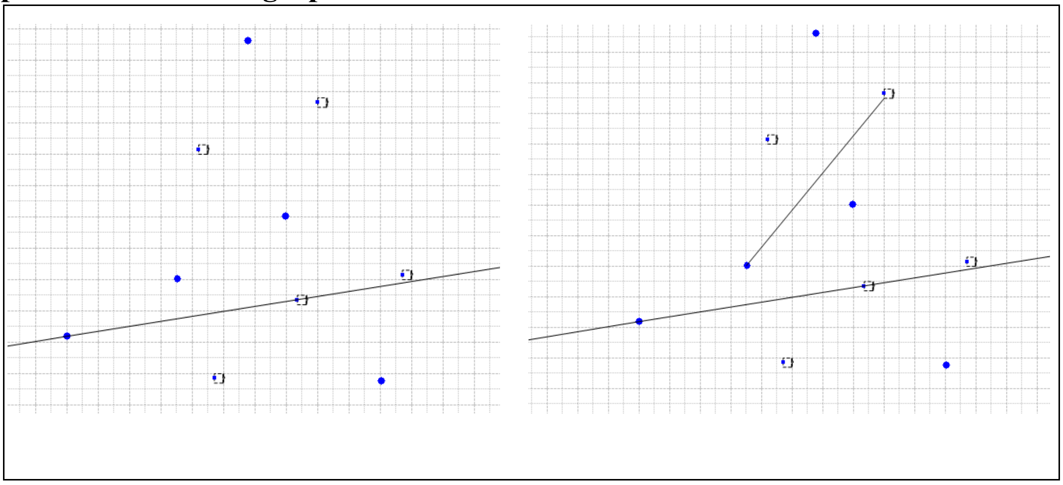
Problem-solving ideas

任意寻找一驱鬼者，与所有鬼相连，必有一条线将战场分为两部分，每部分均有相同数量的驱鬼者与鬼。

Pseudo-code

输入 B 为驱鬼者点集合，G 为鬼的点集合
在 B 中任选一点 B0，与 G0 连接，若 B 和 G 在 B0G0 两侧数量不同，则与 G1 连接…
若数量分别相同，取线段 B0G0，将 B 与 G 分为两部分 D1，D2，F1，F2，继续切分

Subproblem reduction graph



Prove the correctness

使用数学归纳法，记 N 为总点数，显然 N 的取值是偶数

当 N=0 时，显然该算法是正确的。

当 N=2 时，容易验证该算法是正确的。

假设当 0<=N=<=2k 时，算法是正确的，则当 N=2k+2 时，驱鬼者必然可以射中一只鬼，使战场分为两个部分，每部分 0<=N<=2k，算法正确。

因此算法成立。

Analyse the complexity

从任一驱鬼者找到合适的鬼花费时间复杂度 $O(n^2)$，因此总时间复杂度是 $O(n^2 logn)$

附：

分治法

1. 查找数组中第K大元素。在长度为n的无序数组中找到第k大的数。

解法一：对数组进行排序。快速排序。

时间复杂度 $O(n\log n)$

解法二：多次查找。第一遍找到最大值，标记一下，第二遍找到次大值，标记一下，……

时间复杂度：$O(n\cdot k)$

解法三：选择排序、冒泡排序。数据不用全排，只需要排出前K个即可。原理类似于解法二。

时间复杂度：$O(n\cdot k)$

解法四：分治法，借用快速排序的思想。选择一个pivot，通过交换，使得pivot左边的数都大于pivot，右边的数都小于pivot。

若pivot的下标=K-1，则pivot就是第k大的数

若pivot的下标>K-1，则递归地在左边找。

若pivot的下标<K-1，则递归地在右边找。

K=3

K=2

伪代码：

```
getMaxK( int a[], int n, int K ){
    int pos = partion( a, n );
    if ( pos == K-1 ){
        return a[pos];
    }
    if ( pos > K-1 ){
        getMaxK( a, pos, k )        返回2. 排行2大元素
    }
    if ( pos > K-1 ){
        getMaxK( a+pos+1, n-pos-1, K-pos-1 )        返回2. 排行4大元素
    }
}
```

第4大元素，应该得2.

K=2

pos=3

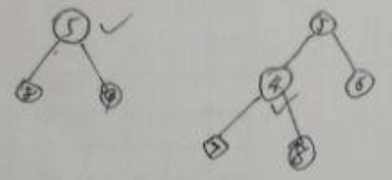
a+3，n-(2+1)

排……的第1大元素。

正确性：用pivot划分之后，第K大元素一定在某一边或pivot本身，故一定可以找到。

复杂度：最好 $O(n)$，最坏 $O(n^2)$，平均近似 $O(n)$。

2. 找一棵完全二叉树的局部最小值。



局部最小值：比与它相邻的结点的值小。

算法描述：从根结点出发，若根结点的值比左右子结点的值都小，则返回根的值。否则，必有一个节点值比根小，递归地走这个结点。

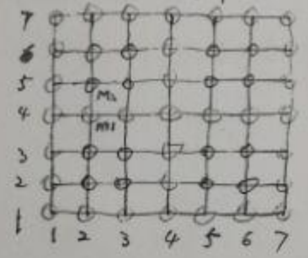
伪代码：

```
FindLocalMin( Tree T){
    if(T->data < T->lchild->data && T->dat < T->rchild->data){
        return T->data;
    }else if( T->lchild->data < T->data){
        FindLocalMin(T->lchild)
    }else{
        FindLocalMin(T->rchild)
    }
}
```

正确性：若根结点比左右子结点小，则根结点的值即为局部最小值。
否则，子结点必有一个小于根结点。这个结点只需与它的子结点比即可。
最坏时遍历到叶子结点，因为比其父结点小，也一定是局部最小值。
∴算法正确。

时间复杂度：O(log n)．

3. 找一个 $n \times n$ 的格子图的局部最小值.



算法描述：考查 $x=1$, $x=7$, $x=4$, $y=1$, $y=7$, $y=4$ 这6条线的交点，并找出最小值的点 m.

这6条线将区域划分为 A_1, A_2, A_3, A_4 四个小区域.

若 m 比周围4个点都小，则 m 就局部最小值.

若 m 旁边有一个数 n，n 的值比 m 小，则在 n 所有的小区域中找.

伪代码：

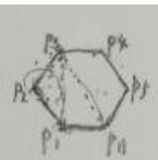
```
FindLocalMin(G){
    m=找到6条线的交点中的最小点;
    if( m < 周围的点){
        return m;
    }else if(m > 大于周围点n){
        FindLocalMin(n点所在的图Gi)
    }
}
```

正确性：一个网格图中必存在局部最小值。在所有边点中，若最小值为 m_1，若 m_1 比周围四个点都小，则 m_1 必是局部最小值. 若 m_1 比 m_2 大，则 m_2 必一定比左上四分之一网格的也都小，则左上网格一定存在局部最小值.

复杂度：

$T(n) = T(\frac{n}{4}) + c$

$\therefore O(n)$.



4. 求凸多边形的三角划分.

问题. 一个凸多边形可以通过若干条互不相交的对角线划分成若干三角形.

四边形有2种划分.

$n=5$ $f=5$

输入边数 n, 求有多少种划分方式.

分析: 因为凸多边形的任意一条边必属于某一个三角形, 所以以某一边为基准, 编号为 P_1、P_n, 其余点编号为 $P_2, P_3, \cdots P_{n-1}$。任取一点 P_k ($k=2,3,\cdots n-1$), 将 P_1, P_k, P_n 将凸多边形划分为由 $P_1, P_2, \cdots P_k$ 组成的凸k边形 A, 由 $P_k, P_{k+1}, \cdots P_n$ 组成的凸 $(n-k+1)$ 边形 B. 和三角形 $P_1P_kP_n$.

当k确定时, 总划分数 = A的划分总数 × B的划分数 $= f(k)\times f(n-k+1)$.

伪代码:

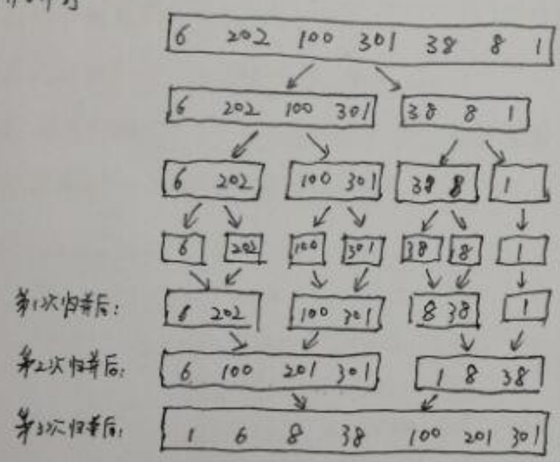
```
SEPARATED(n) {
    sum=0;
    if(n≤3) return 1;
    for( k=2, k<n; k++) {
        sum = sum + SEPARATED(k) * SEPARATED(n-k+1);
    }
    return sum;
}
```

正确性分析: 根据分析, 正确.

复杂度: 每个n的问题会分成n-1个子问题, 每个子问题包含n-1个乘法.

$\therefore O(n^2)$

5. 归并排序




```
MergeSort(A, left, right){
    if(left < right){
        mid = (left+right)/2;
        MergeSort(A, left, mid);
        MergeSort(A, mid, right);
        Merge(A, left, mid, right);   //合并两个有序数组.
    }
}
Merge(A. left, mid, right){
    int i=0 ,j=0 ,k=0;
    ~~for(k=left; k≤right; k++)~~{
        ~~if(L[i] ≤ R[j]){~~
            ~~A[k]=~~
    while(i≤mid && j≤right){
        if(A[i] < A[j]){
            temp[k++] = A[i++];            //temp 自行处理.
        }else{
            temp[k++] = A[j++];
        }
    }
    while(i≤mid){
        temp[k++] = A[i++];
    }
    while(j≤right){
        temp[k++] = A[j++];
    }
}
```

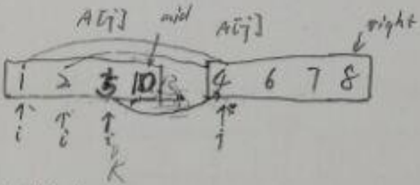
left L: mid　　R: right

1 3 7 9　　2 4 5 8

i　　j

$T(n) = O(n\log n)$

6. 求数组中逆序对个数.



学习的思想，排序

方法一：利用归并~~的思想~~，在归并的过程中

若 A[i] < A[j]，则~~逆序~~不是逆序数，i++

若 A[i] > A[j]，则 A[i] 到 A[mid] 都与 A[j] 构成逆序数，j++.

可以在归并的过程中求出逆序数.

代码：
```
MergeSort1 ( A , left, right) {
    if( left < right ) {
        mid = (left +right)/2 ;
        MergeSort1 ( A, left, mid);
        MergeSort1 (A, mid, right);
        Merge ( A, left, mid, right);
    }
}
Merge1 ( A, left, mid, right) {
    int i=0, j=0, K=0;
    while ( i≤mid && j≤right) {
        if( A[i] < A[j] ){
            temp[K++] = A[i++];
        }else{
            sum = sum + mid-i+1;
            temp[k++] = A[j++];
        }
    }
    while ( i≤mid ){
        temp[K++] = A[i++];
    }
    while ( j≤right ){
        temp[K++] = A[j++];
    }
}
```

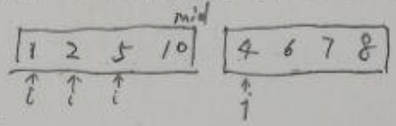
就比归并排序多了一句话.

T(n) = O(n log n)

7. 求数组中另一种"逆序对"个数.

定义逆序对：若 i<j，当 ai > 3aj 时 (ai, aj) 才称为逆序对. 如 (5,2) 不是，(6,2) 不是，(7,2) 是.

算法：求一般的逆序数时，Merge时，



当 A[i] > A[j] 时，A[i] 之后的元素都将大于 A[j] 构成逆序数.

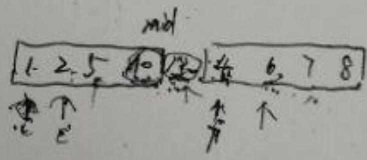
而这里有的时不满足，可以构造一个单独的函数计算两个有序数组中的逆序数.

代码：

```
MergeSort2(A, left, right){
    if(left < right){
        mid = (left+right)/2;
        MergeSort2(A, left, mid);
        MergeSort2(A, mid, right);
        Merge2(A, left, mid, right);
    }
}

Merge2(A, left, mid, right){
    int i=0, j=0, k=0;
    Count(A, left, mid, right);
    while(i≤mid && j≤right){
        if(A[i] < A[j]){
            temp[k++] = A[i++];
        }else{
            temp[k++] = A[j++];
        }
    }
    while(i≤mid){
        temp[k++] = A[i++];
    }
    while(j≤right){
        temp[k++] = A[j++];
    }
}
```

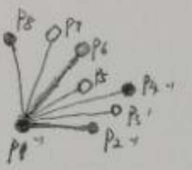
比归并排序多了一个函数.



```
Count(A, left, mid, right){
    i=0, j=0;
    sum
    while(i≤mid && j≤right){
        if(A[i]≤3A[j]){
            i++;
        }elses{
            sum=sum+mid-i+1;
            j++;
}   }   }
```

8. 巨人与鬼

问题：平面中有n个黑点和n个白点，要求画n条线段，每条线段连接一个黑点和一个白点。要求n条线段不相交。已知这2n个点没有3个点共线。



(1). 在 $O(n\log n)$ 时间内找出这样一条线段，将平面分成两个半平面且每个半平面中黑点数等于白点数。

算法：首先找到最下，最左的点 P_1，其他点按相对于 P_0 的极角由小到大编号 $P_2, P_3 \cdots P_n$

设鬼的标记为-1，巨人的标记为1。

从 P_2 开始扫描，当扫到某点时，扫过的点标记之和为0时，说明扫过的巨人和鬼的数目相同，

如本例，$P_1+P_2=-2$，$P_1+P_2+P_3=-1$，$P_1+P_2+P_3+P_4=-2$，$P_1+P_2+P_3+P_4+P_5=-1$，$P_1+P_2+P_3+P_4+P_5+P_6=0$

则找到线段 P_1P_6 满足条件。

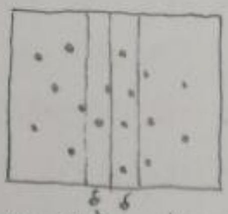
(2) 给出 $O(n^2\log n)$ 的算法找出n条线段。

利用分治法思想，首先根据(1)中算法划分，然后左右两边分别递归求。

```
void Solve( left , right ){
    找到最左下角节点，编号为Pl;
    计算其他节点与Pl的角度，按角度排序，编号为Pl+1,Pl+2,…Pr.
    for(i=l, i<=r, i++){
        n+=P[i].type ;
        if(n==0){
            输出连线 PlPi,
            Solve( left+1, i-1);
            Solve( i+1, right);
            break;
        }
    }
}
```

9. 最近点对问题.

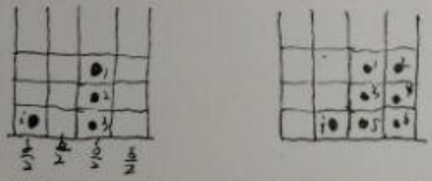
分析：



采用分治法思想，将点按照X坐标分成两部分。最近点对可能存在于各部分中，也可能一个点在左边，一个点在右边。求左右的最近点可通过递归完成，设 min(左,右)=δ
在划分处取 X±δ 的长条，否则最近点对距离为δ，每个点都在长条中取。
对长条内的点按y坐标排序，每个点只需要跟其后的11个点比较即可。

伪代码：
```
ClosestPair(Pi, P2, ... Pj){    // Pi ... Pj 已经按X坐标排序
    if(j-i == 1){
        return d(Pi, Pj);
    }
    用 P(i+j)/2 的X坐标将 Pi ... Pj 划分成两部分.
    δ1 = ClosestPair(左半部分);                      T(n/2)
    δ2 = ClosestPair(右半部分);                      T(n/2)
    δ = min(δ1, δ2)
    对 X-2δ ~ X+2δ 内的点按y坐标排序，             O(nlogn)
    对于排序好的点，计算每个点与其后的11个点的距离，若比δ小，则更新δ.   O(n)
}
```

正确性证明：



(1) 将长条划分为边长为 $\frac{\delta}{2}$ 的小格，则每个小格内最多只有1个点（若有多个，则最短距离小于δ）
(2) 对于点i，只需检查1,2,3这三点即可（其他点距离i一定大于δ）.
(3) 对于点i，只需检查1,2…6这6个点即可.
综上，若对y坐标排序，只须检查本行与后两行即可，只需检查11个点。算法正确.

时间复杂度： $T(n)=2T(\frac{n}{2})+O(n\log n)=O(n\log^2 n)$