1.COVID-19 Risk Detection

(a)

伪代码

```
输入:
A:数组
  A[i][0]表示第i个人进店的时间
  A[i][1]表示第i个人离店的时间
Ci:第Ci个人被判为确诊
输出:有可能被感染的人的数组

Get_Potencial_Inflicted_Customers(A,Ci)
  out = []
  k = 0
  for i = 0 to A.length-1:
    if A[i][0] > A[Ci][1] or A[i][1]<A[Ci][0] or i = Ci:
        continue
    else
        out[k++]=i
  return out
```

> 一次遍历，时间复杂度为$O(N)$

(b)

伪代码

```
输入:
A:数组
  A[i][0]表示第i个人进店的时间
  A[i][1]表示第i个人离店的时间
输出:在同一时间出现在店里的客人的有序对数

Get_Same_Time_Pairs(A)
  answer = 0
  for i = 0 to A.length-1:
    temp = 0
    for j = i + 1 to A.length-1:
      if A[i][0] > A[j][1] or A[i][1] < A[j][0]:
        continue
      else
        temp++
    answer = answer + temp
  return answer
```

> 两次遍历，时间复杂度为$O(N^2)$

(c)

伪代码:

```
输入:
A:数组
  A[i][0]表示第i个人进店的时间
  A[i][1]表示第i个人离店的时间
输出:在同一时间出现在店里的客人的有序对数

Get_Same_Time_Pairs(A)
 Quick_Sort(A,0,A.length-1)
  sum = 0
  for i = 0 to A.length-1
    index = Binary_Search(A,A[i][1])
    sum += index - 1
  return sum


Binary_Search(arr,target)//二分查找小于target的最大值
  left = 0
  right = arr.length-1
  while left < right
    middle = left + (right - left)/2
    if arr[middle] = target
      return middle
    else if arr[middle] < target
      left = middle
    else
      right = middle - 1
  return left

Quick_Sort(A,p,r)//快排
 if p < r
   q = Partition(A,p,r)
   Quick_Sort(A,p,q-1)
   Quick_Sort(A,q+1,r)

Partition(A,p,r)
 x = A[r]
 i = p - 1
 for j = p to r - 1
   if A[j] <= x
     i = i + 1
     exchange A[i] with A[j]
 exchange A[i+1] with A[r]
 return i + 1
```

> 1. 首先将A数组根据进店的时间快速排序($O(nlogn)$)
> 2. 遍历A到第i个人，利用A数组二分查找进店时间在A[i][1]前的人(二分:$O(logn)$)
> 3. 每次遍历结果加上返回的最大的离店时间小于A[i][1]的客人的序号减去i
> 4. 总的时间复杂度$O(nlogn) + nO(logn) = O(nlogn)$

2.Proof of correctness

第一层循环不变式:

初始化:

第一次循环前:i=0

保持:

每次循环前A[i]前面的数组都是已经排好序了的，接着获取到A[i]...A[A.length]中的最小值的位置，然后与A[i]交换，保证了A[0]...A[i]的所有元素都排好序

终止:当i = A.length时，终止循环。完成排序

第一层循环不变式:

初始化:

第一次循环前，minIndex=i, j = i + 1

保持:

每次循环前，minIndex的值都是A[i]...A[j-1]中最小值的位置，然后循环判断当前值是否小于A[minIndex]的值，如果是的话就更新minIndex，这样就保证了每次循环都会使minIndex记录A[i]...A[j]中最小值的位置

终止:

当j = A.length+1时，循环结束，minIndex记录了A[i]...A[A.length]中的最小值的位置。

3.Needlessly complicating the issue

(a)

伪代码

```
Find_Minimun(A)
  min = A[0]
  for i = 1 to A.length-1
    if A[i] < min
      min = A[i]
  return min
```

> 一次遍历，记录最小值，时间复杂度为$O(n)$

(b)

因为要从n个数中选出最小的数，所以每个数至少需要参与一次比较，所以无论什么算法都必须至少n次操作

(c)

1.

A[0]

2.

findMinimum(A)的功能就是找到A数组中的最小值

通过将0...n分为A_1,A_2两个数组，通过findMinimum(A_1)和findMinimum(A_2)分别找出A_1, A_2中的最小值a_1, a_2，然后返回$min(a_1, a_2)$，就能返回A的最小值，中途过程是采用的递归的方式，直到数组中只有1个数时才停止递归，直接返回这个数。

(d)

a题算法中，一共需要n-1次比较，最好情况下1次赋值，最坏情况n次赋值

b题算法中，若$n = 2^k$，则需$1 + 2 + 4 + ... \frac{n}{2} = n - 1$次比较，每一次递归调用函数除了叶子节点以外都需要2次赋值，共$2(1 + 2 + 4 + ... + \frac{n}{2}) = O(n)$次赋值

4.Recursive local-minmum-finding

(a)

(1)

伪代码

```
Find_A_Min_Loc(A)
  if A.length = 1
    return A[0]
  if A.length = 2
    return min(A[0],A[1])
  mid = A.length/2
  if A[mid] < A[mid-1] and A[mid] < A[mid+1]
   return A[mid]
  else if A[mid] > A[mid-1]
   return Find_A_Min_Loc(A[0:mid-1])
  else
   return Find_A_Min_Loc(A[mid+1:A.length])
```

(2)

定理一:**一个不同数组成的数组一定存在一个局部最小值**

证明，若不存在局部最小值那么肯定$a_2 < a_1$，a_3也一定要比a_2小，不然a_2就是局部最小只了，同理下去$a_2 < a_1, a_3 < a_2, a_4 < a_3, \ldots, a_n < a_{n-1}$,而$a_n < a_{n-1}$，就代表$a_n$就是局部最小值，因此矛盾

定理二：**取数组中间的数，如果它不是局部最小值，那么在小于它的邻数那边肯定存在局部最小值。**

证明:和定理一的证明相似若$a_{mid} > a_{mid+1}$，若要不存在局部最小值话，就会推出$a_n > a_{n-1}$，那就会与a_{n-1}为局部最小值产生矛盾，因此小于它的邻数那边肯定存在一个局部最小值。

有了以上两条推论我们就可以证明算法的正确性了

初始化:

当A的长度为1时，直接返回唯一元素，为局部最小值；当A的长度为2的时候，返回两个数中更小的那一个，即为局部最小值。

保持:

当A的长度大于3的时候，取数组A的中间元素，然后判断中间元素是否为局部最小值，若是则直接返回中间元素。如果不是，则选取相邻数中小于它的一边。然后再递归求取这一半数组中的局部最小值，根据定理二我们知道肯定有这样一个局部最小值，因为数组A,一定能在Find_A_Min_Loc(A[left]),A[mid],Find_A_Min_Loc(A[right])中找到局部最小值。因此最后便会返回数组A的局部最小值。

结束

当递归返回到父节点后算法结束，返回数组A的局部最小值。

(3)

$O(T_n) = O(T_{\frac{n}{2}}) + C$，所以时间复杂度为$O(logn)$

(b)

(1)

伪代码

```
GET_MIN_LOC(A,i,j,direct)
 min_x,min_y = FIND_MIN_LINE(A,i,j,direct)
 if direct == 0 //横线
   if A[min_x][min_y] < (min_x-1 < 0) ? MAX_VALUE : A[min_x-1][min_y] and
A[min_x][min_y] < (min_x+1 >=         A.length-1) ?
MAX_VALUE : A[min_x+1][min_y]
     return A[min_x][min_y]
   else if min_x+1 >= A.length or A[min_x-1][min_y] < A[min_x+1][min_y]
     return GET_MIN_LOC(A[0:min_x,0:-1],min_x,min_y,(direct+1)%2)
   else if min_x-1 < 0 or A[min_x+1][min_y] < A[min_x-1][min_y]
     return GET_MIN_LOC(A[min_x:-1,0:-1],min_x,min_y,(direct+1)%2)
 else if direct == 1 //竖线
   if A[min_x][min_y] < (min_y-1 < 0) ? MAX_VALUE : A[min_x][min_y-1] and
A[min_x][min_y] < (min_y+1 >         A[0].length-1) ? MAX_VALUE :
A[min_x][min_y+1]
     return A[min_x][min_y]
```

```
  else if min_y+1 >= A[0].length or A[min_x][min_y-1] < A[min_x][min_y+1]
    return GET_MIN_LOC(A[0:-1,0:min_y],min_x,min_y,(direct+1)%2)
  else if min_y-1 < 0 or A[min_x][min_y+1] < A[min_x][min_y-1]
    return GET_MIN_LOC(A[0:-1,min_y:-1],min_x,min_y,(direct+1)%2)

FIND_MIN_LINE(A,i,j,direct)
 if direct == 0 //横线
   int min_j = 0
   for k = 1 to A[0].length-1
     if A[i][k] < A[i][min_j]
       min_j = k
   return i,min_j
 else if direct == 1 //竖线
   int min_i = 0
   for k = 1 to A.length-1
     if A[k][j] < A[min_i][j]
       min_i = k
   return min_i,j
```

(2)

算法思路:

> 先在正中间画一条横线,找到横线上最小的位置a。
>
> 如果这个位置上下两个位置的数(a_1,a_2)都比它大，那么它是局部最小。
>
> 如果上下两个位置的数都比它小，**那么随便舍弃哪一边的矩阵都行(都能保证能找到局部最小解，但保留矩阵内元素个数更少的那个矩阵肯定会搜索更快)**
>
> 如果上下两个位置的数只有一个比它小，那就保留上下两个数中数小的所在的那边的矩阵。
>
> 然后再在这个保留下来的矩阵中作一条通过这个数的竖线，之后的思路与上面过程一样，不断的横线竖线，将矩阵减小，直到找到局部最小值。

正确性证明:

定理一:**一个每个元素都不同的矩阵中至少存在一个局部最小值**

证明，因为每个数都不同，所以肯定有一个最小值，那它肯定是局部最小值。

定理二:**每次选择比横(竖)线上最小值更小数所在的那边的矩阵肯定能找到局部最小值**

这句话可能比较绕，举个例子，如果横线上的最小值是a_{ij}，那么若$a_{ij+1} < a_{ij}$，那选取a_{ij+1}那边的矩阵，肯定能找到局部最小值。同理若$a_{i+1j} < a_{ij}$也是这样，为什么呢?

因为根据定理一，选取的那个矩阵中肯定存在一个局部最小值，并且肯定不在那条横线上。因为如果在那条横线上，而又有$a_{ij+1} < a_{ij}$,说明它不是局部最小值那就肯定矛盾了。所以一定在内部

所以，根据以上两个定理，我们的算法一定能通过每次的划分而找到一个局部最小值。

(3)

每两次画的线长度会减半，画的线总长度是$O(n)$

因为每次矩阵切分的时候都会平均减去n/2的矩阵大小，所有搜索的元素大概是 $n + n/2 + n/4 + n/8 + ... = 2n = O(n)$

5 Probability refresher

(a)

$C_n^0 + C_n^1 + C_n^2 + ... + C_n^n = 2^n$

> {1,2,...,n}的所有子集中包含0个元素的个数为为C_n^0，包含一个元素的为C_n^1，...包含n个元素的为C_n^n,它们加起来总共就有2^n个.

(b)

$\sum_{i=0}^{n} i * \frac{C_n^i}{2^n} = \frac{n}{2}$

(c)

得出的范围是(k, kn)

平均分布的期望是$\frac{a+b}{2}$,方差是$\frac{(b-a)^2}{12}$

所以expected value是$\frac{(n+1)k}{2}$,variance是($\frac{(kn-k)^2}{12}$)

6 Fun with Big-O notation

$(a) n = O(nlog(n))$

答:设当$n >= n_0$时，存在c_2，使$n <= c_2 * nlogn$

$\Rightarrow 1 <= c_2 * logn$

$\Rightarrow c_2 >= \frac{1}{logn}$

则取$c_2 = 1, n_0 = 2$，则当$n >= 2$时，$c_2 >= \frac{1}{logn}$始终成立

故$n = O(nlog(n))$为真

$(b) n^{1/logn} = \Theta(1)$

答:设$logn = k => n = 2^k$,

则$n^{1/logn} = (2^k)^{\frac{1}{k}} = 2$

所以令$c_1 = 1, c_2 = 3, n_0 = 1$，当$n >= n_0$时，恒有$c_1 * 1 <= n^{1/logn} <= c_2 * 1$，故 $n^{1/logn} = \Theta(1)$为真

$(c) if$

$$f(n)=\begin{cases}5^n & n<2^{1000}\\ 2^{1000}n^2 & n>=2^{1000}\end{cases}$$

$and\ g(n)=\frac{n^2}{2^{1000}}, then f(n)=O(g(n))$

答:设当$n\geq n_0$时，存在c_2使

$f(n)<=c_2*g(n)$

$\Rightarrow f(n)\leq c_2*\frac{n^2}{2^{1000}}$

$\Rightarrow c_2\geq \frac{2^{1000}}{n^2}*f(n)$

则当$n_0=2^{1000}$时,

$c_2\geq \frac{2^{1000}}{n^2}*2^{1000}*n^2$

$\Rightarrow c_2\geq 2^{2000}$

所以当$c_2=2^{2000}, n_0=2^{1000}$时，恒有$f(n)\leq c_2*g(n)$,故$f(n)=O(g(n))$成立

(d) For all posible fucntions $f(n),g(n)\geq 0$,if $f(n)=O(g(n))$,then $2^{f(n)}=O(2^{g(n)})$

答:由$f(n)=O(g(n))=>$存在n_1,c_1，当$n\geq n_1$时，恒有$f(n)\leq c_1*g(n)$

反证法，假设存在n_2,c_2使得，当$n\geq n_2$时，$2^{f(n)}=O(2^{g(n)})=>2^{f(n)}\leq c_2*2^{g(n)}$恒成立。

由当$n\geq n_1$时，$f(n)\leq c_1*g(n)$，则$2^{f(n)}\leq 2^{c_1*g(n)}\leq c_2*2^{g(n)}=>c_2\geq 2^{(c_1-1)g(n)}$。如果$g(n)$为增函数，那么当$n->\infty, c_1>1$时，$2^{(c_1-1)g(n)}->+\infty$，而$c_2$是一个常数，所以不可能。故为假

$(e) 5^{loglog(n)}=O(log(n)^2)$

答:假设当$n\geq n_0$时存在c_2，恒有$5^{loglogn}\leq c_2 logn*logn$

$\Rightarrow loglogn\leq log_5^{c_2}+log_5^{logn}+log_5^{logn}$

$\Rightarrow log_2^{logn}-2log_5^{logn}\leq log_5^{c_2}$

$\Rightarrow log_2^{logn}-log_{\sqrt{5}}^{logn}\leq log_5^{c_2}$

$\Rightarrow c_2\geq 5^{log_2^{logn}-log_{\sqrt{5}}^{logn}}$

由于$log_2^{logn}-log_{\sqrt{5}}^{logn}$是增函数，所以当$n->+\infty$时，$c_2\geq +\infty$，因为$c_2$是常数，所以肯定不可能。

所以$5^{loglog(n)}=O(log(n)^2$为假

$(f) n=\Theta(100^{log(n)})$

答:假设存在n_0, c_1, c_2，当$n \geq n_0$时，恒有$c_1 * 100^{logn} \leq n \leq c_2 * 100^{logn}$

$\Rightarrow c_1 \leq \frac{n}{100^{logn}} \leq c_2$

令$logn = k \Rightarrow n = 2^k$

$\Rightarrow c_1 \leq \frac{2^k}{100^k} \leq c_2$

$\Rightarrow c_1 \leq (\frac{1}{50})^k \leq c_2$

当$n \rightarrow +\infty$的时候，k$\rightarrow \infty \Rightarrow (\frac{1}{50})^k \rightarrow 0$，因为$c_1$是一个正常数，所以肯定不可能

所以

$n = \Theta(100^{log(n)})$为假

7 Fun with recurrences.

(a)

$T(n) = 2T(n/2) + 3n$

因为a=2,b=2,d=1

所以$a = b^d$

故$O(T) = O(nlogn)$

(b)

$T(n) = 3T(n/4) + \sqrt{n}$

因为$a = 3, b = 4, d = \frac{1}{2}$

所以$a > b^d$

故$O(T) = O(n^{log_4^3})$

(c)

$T(n) = 7T(n/2) + \Theta(n^3)$

因为$a = 7, b = 2, d = 3$

所以$a < b^d$

故$O(T) = O(n^3)$

(d)

$T(n) = 4T(n/2) + n^2 logn$

因为$a = 4, b = 2$且$f(n) = \Omega(n^{log_2^4})$

且$4f(n/2) \leq cf(n)$

$\Rightarrow 4 * \frac{n^2}{4} log\frac{n}{2} \leq cn^2 logn$

$\Rightarrow log\frac{n}{2} \leq clogn$

当$c = 1, n \rightarrow \infty$的时候恒成立

所以$O(T) = O(n^2 logn)$

(e)

$T(n) = 2T(n/3) + n^c$

因为a=2,b=3,d=c

1.当$2 = 3^c \Rightarrow c = log_3^2$

$O(T) = O(n^{log_3^2} log(n))$

2.当$2 < 3^c \Rightarrow c > log_3^2$时

$O(T) = O(n^{log_3^2})$

3.当$2 > 3^c \Rightarrow c < log_3^2$时

$O(T) = O(n^{log_3^2})$

(f)

$T(n) = 2T(\sqrt{n}) + 1, where\ T(2) = 1$

$\Rightarrow T(n) = 2T(n^{\frac{1}{2}}) + 1$

$\Rightarrow 2^{2^k} = n$

$\Rightarrow k = log_2^{log_2^n}$

经过k次递归后，达到叶子节点T(2)

则总的时间复杂度为$O(1 + 2 + 4 + 8 + \ldots 2^k) = O(2log_2^n - 1) = O(logn)$