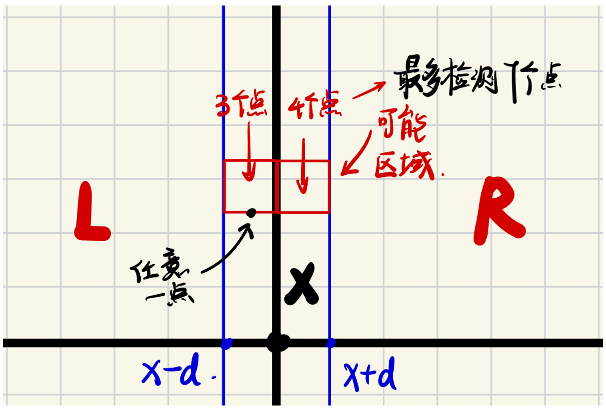
PS3-231830106

Problem 1

(a):

- a.1: 算法将点分为L和R两部分，先通过分治(递归)对L和R分别求对近点对 (p_L, q_L, d_L)和(p_R, q_R, d_R)，设置$d = min\{d_L, d_R\}$再通过限制条件检验跨区点对中是否存在$d_M < d$，$result = min\{d_L, d_R, d_M\}$
- a.2: 只需证明递归步骤对正确性即可，每一步中：首先分别可以递归找到L和R中的最近点对，然后检测跨区对中的点，检测在x附近宽度为2d的区间中的点，已知：在$d \times d$的正方形中最多有4个点，因此只需检查一个$d \times 2d$的区域中的点，而这个区域中最多最多只有7个点。
  附图:
  

  因此只需要计算后续7个点和原点点距离并与d比较即可。
- a.3递归每一步正确则显然算法整体正确。

(b):

1. 伪代码：

```
function ClosestPair(points):
        // 递归终点
        if n <= 3:
                return findclosest(points)

        // 按x和y大小顺序排序
        sorted_by_x = sort (points,by"x")
        sorted_by_y = sort (points,by"y")

        //分为L和R两个部分
        x_mid = findmiddle(sorted_by_x)
```

```
        L = sorted_by_x[0,n/2]
        R = sorted_by_y[n/2,n]

        //递归求解
        (p_L,q_L,d_L) = ClosestPair(L)
        (p_R,q_R,d_R) = ClosestPair(L)
        d = min(d_L,d_R)

        //检测跨区对
        strip = [p for p in points if |p.x-x_mid|<d]
        (p_M,q_M,d_M) = strip_ClosestPair(strip)

        //返回答案
        return cloest_of((p_L,q_L),(p_R,q_R),(p_M,q_M))

function findmiddle(sorted_by_x):
        return (sorted_by_x[len(sorted_by_x)/2]).x

function strip_ClosestPair(strip):
        (p_M,q_M,d_M)=(0,0,0)
        n = len(strip)
        for i in range(n):
                for j in range(i,min(i+7,n)):
                        d = distance(strip[i],strip[j])
                        if d < d_M:
                        (p_M,q_M,d_M)=(strip[i],strip[j],d)
        return (p_M,q_M,d_M)

function findclosest(points):
        if len(points) == 2:
                return points
        return cloest((points[0],points[1]),(points[1],points[2]),
(points[0],points[2]))
```

2. 时间复杂度：

- 根据分治方法: $T(n)=2T(n/2)+O(n\log n)$
- 解释：每次递归将问题分为两个子问题，因此有$T(n)=2T(n/2)$；在每层递归中，需要对x和y做一次排序，其时间复杂度为$O(n\log n)$则总递归式为 $T(n)=2T(n/2)+O(n\log n)$
- 根据主定理：时间复杂度为$TO(n\log^2 n)$

(c):

- 解析：只需要在接收输入时将points先进行一次排序即可，这样避免了过程中的重复排序，使得递归变成$T(n)=2T(n/2)+O(n)$。
- 伪代码：

```
function ClosestPair(sorted_by_x, psorted_by_y):
    if n <= 3:
        return findclosest(points_sorted_by_x)

    x_mid = find_middle(sorted_by_x)

    L_x = sorted_by_x[0:n/2]
    R_y = sorted_by_x[n/2:n]

    L_y = [p for p in sorted_by_y if p.x <= x_mid]
    R_y = [p for p in sorted_by_y if p.x > x_mid]

    (pL, qL, dL) = ClosestPair(L_x, L_y)
    (pR, qR, dR) = ClosestPair(R_x, R_y)

    d = min(dL, dR)

    strip = [p for p in sorted_by_y if |p.x - x_mid| < d]
    (pM, qM, dM) = strip_ClosestPair(strip, d)

    return closest_of(pL, qL, pR, qR, pM, qM)
```

- 递归关系: $T(n)=2T(n/2)+O(n)$,由主定理可得时间复杂度为$O(n\log n)$

Problem 2

(a):

1. 思路：插入元素后与旧元素比较，进行上浮和下沉操作即可。
2. 伪代码

```
function HeapUpdate(heap,i,val):
        // 更新索引对应的值
        old_val = heap[i]
        heap[i] = val
```

```
        //上浮
        if val > old_val:
                Heap_up(heap,i)

        if val == old_val:
                return 0
        //下沉
        else:
                Heap_down(heap,i)

function Heap_up(heap,i):
        while i>1 and heap[i]>heap[parent(i)]:
                swap(heap[i],heap[parent[i]])
                i = parent(i)



function Heap_down(heap,i)
        while (heap[i] < heap[left(i)] and left(i) < size(heap) )or
(heap[i] < heap[right(i)] and right(i) < size(heap)):
        if heap[left(i)] < heap[right(i)]:
                swap(heap[i],heap[right[i]])
                i = right[i]
        else:
                swap(heap[i],heap[left[i]])
                i = left[i]

function parent(i):
        return i//2
function left(i):
        return i*2
function right(i):
        return i*2+1
```

3. 时间复杂度：heap最大层数为$\lg n$，而每次操作顺着树的枝移动，因此操作次数不会大于$\lg n$，时间复杂度为$O(\lg n)$

(b):

1. 利用一个辅助最大堆，先将最大堆的堆顶插入辅助最大堆，然后每次将辅助最大堆的堆顶的元素取出，并将在原最大堆中的子节点插入辅助最大堆中，重复k次即可。
2. 伪代码

```
function Find_k_heap(heap,k):
        maxheap = Max_heap()
        Insert(maxheap,heap[1],1)

        for i in range(1,k+1):
        (max,id) = getmax_heap(maxheap)

        if leftchild(id) <= n:
                Insert(maxheap,heap[leftchild(id),leftchild(id)])
        if rightchild(id) <= n:
                Insert(maxheap,heap[rightchild(id),rightchild(id)]

        return max

leftchild(i):
        return i*2
rightchile(i):
        return i*2+1
```

3. 时间复杂度: 在辅助最大堆中进行k次维护，每次维护时堆的层数不超过$\lg k$，因此时间复杂度不超过$O(k\lg k)$

Problem 3

1. 思路：将每个链表的表头的值放入一个最小堆中，每次取出最小堆的堆顶元素，并将取出元素的链表表头指向下一个元素，更新最小堆，重复n次即可依次取出最小值。
2. 伪代码：

```
function Heaplist(lists):
        //定义最小堆
        min_heap = new Minheap()

        //将每个链表的表头放入最小堆中
        for list in lists:
                if list is not null:
                        Insert_heap(list.val,list)

        //创建一个虚拟头节点
        dummy = new Listnode(o)
        current = dummy

        //取出最小元素构建链表
```

```
        while not min_heap.isEmpty():
                //取出最小元素
                min, node = min_heap.extractmin()
                //添加到链表中
                current.next = node
                node = currrent
                //将下一个节点插入堆
                if node.next is not null:
                        Insert_heap(node.next.val,node.next)
        //返回合并后链表
        return dummy.next
```

3. 时间复杂度：

- 最小堆共有k个元素，因此每次堆的插入和删除时间复杂度都为$O(\lg k)$，共有n个元素，每个元素入堆一次出堆一次，只有常数级堆操作，因此总时间复杂度为$O(n\lg k)$。

Problem 4

1. 思路：先用归并排序对x对值进行排序，再线性扫描排序后的区间数组，将相邻两个区间合并，得到的数组即为答案。
2. 伪代码：

```
function mergeIntervals(intervals):
        //对x排序
        interval_sorted_by_x = sort(intervals,by"x")

        //初始化结果
        merged = []

        //扫描数组
        i = 0
        while i <= len(intervals):
                a = interval_sorted_by_x[i]
                b = interval_sorted_by_x[i+1]
                if i == len(intervals) is empty or a.end <= b.start:
                        merge.append(a)
                        i++
                else:
                        b = Merge_intervals(a, b)
                        i++
```

```
        return merged
```

3. 时间复杂度：

- 第一次归并排序时间复杂度为$O(n\lg n)$，扫描数组时间复杂度为$O(n)$，显然总时间复杂度为$O(n\lg n)$。

Problem 5

(a):

1. 算法简述：找到现在部分最大的pancake将其翻转到顶部，再将全部pancake翻转，这样就能将最大的pancake放在底部。接着对除了最底部的最大盘子的剩下部分做上一步操作。这样只需要2n步操作就能完成。只需要$O(n)$次翻转。
2. 伪代码

```
function flipsort(pancakes,n):
        if n==1:
                return pancakes

        max_idx = findmax(pancakes,n)

        if max_idx != 0:
                flip(pancakes, max_idx+1)
        flip(pancakes,n)

        return flipsort(pancakes,n-1)
```

3. 正确性和复杂度：

- 正确性：每次flipsort都确定的将最大的pancake放在了最底部，因此n次flipsort后所有的pancake必然已经排好序。
- 复杂度：每次flipsort需要至多两次翻转，flipsort共n次，显然只需要$O(n)$次翻转。

(b):

1. 构造：

-
  ```
  ```

```
len(pancake) = n mid = n//2+1
pancakes = [n,mid-1,n-1,mid-2,n-2,……,mid]
```

3. 复杂度：每次flipsort需要至少2次翻转，总翻转次数显然为$\Omega(n)$。

Problem 6

(a):

- $T(n)=3T(2n/3)$，由主定理可得时间复杂度为$O(n^{\log_{3/2}3})$

(b):

- 考虑每一次递归中的三次排序($m=\lceil 2n/3\rceil$)：
- 认为m将A分为L,M,R三段
  1. 将前m个元素排序，此时max(L)<=min(M)
  2. 将后m个元素排序，此时R段已经排序好且min(R)>=max(M).
  3. 再次将前m个元素排序,显然排序前max(M)<min(R),假设在排序前存在max(L)>min(R),而min(R)在后m个元素中比m/2个元素大，而在第一次排序后已经有max(L)<min(M),即max(L)比后m个元素中至少m/2个元素小，显然矛盾 ==> max(L)<=min(R),又max(M)<=min(R)，可得后n-m个元素已经排序好，第三次排序又将前m个元素排序好，且后n-m个元素的的最小值比前m个元素的最大值大。
  4. 即证LMR分别排序好，且有max(L)<=min(M),max(M)<=min(R)

(c):

- 将m从$\lceil 2n/3\rceil$调整为$\lfloor 2n/3\rfloor$仅仅改变了M段的长度，不影响证明，因此原算法仍然正确。

(d):

- $A[i]$和$A[j]$(0<=i<j<=n-1)如果进行了一次交换其大小关系则已经排好序，逆序对数量-1，显然在三次排序中n序对的数量严格递减，如果存在$A[i]$和$A[j]$的第二次交换，则逆序对数量增加，矛盾。因此任意两个A中的元素至多交换一次，则交换次数为$O(\binom{n}{2})$。
-