DSA HW #2

Problem 1

Subproblem 1:

照拿到順序去問鬆餅神，每次被判斷成中間的口味就換成下一個
直到最後三個再問一次，不是中間的兩個就是boundary

```
Function Find_Boundary(P[n])
    i, j, k = 0, 1, 2
    for a from 4 to n:
        median = Pancake-God-Oracle(P, i, j, k)
        if P[i] = median
            i = a
        else if P[j] = median
            j = a
        else if P[k] = median
            k = a
    median = Pancake-God-Oracle(P, i, j, k)
    if P[i] = median
        return j, k
    else if P[j] = median
        return i, k
    else if P[k] = median
        return i, j
    end if
```

Subproblem 2:

ref: https://alrightchiu.github.io/SecondRound/comparison-sort-merge-sorthe-bing-pai-xu-fa.html
用類似merge sort的方法，先找出兩個boundary並取其中一個當作最後點 c 和從中間取出的兩個鬆餅 a b 做比較。如此吐出來的median a 必放在 b 之後，因此可拿來做merge sort

```
a , b = Find_Boundary(P)
Function Merge(P,a,buffer1,buffer2,front,mid,end):
    idx1 , idx2 = 0 , 0
    Copy(P[front,front+1,...mid] , buffer1)
    Copy(P[mid+1,mid+2...end] , buffer2)
    for i=front i<=j i+=1:
        if Pancake-God-Oracle(a,front,mid+1)==front:
            P[i] = buffer[idx1]
            front+=1
            idx1+=1
        else:
            P[i] = buffer[idx2]
            mid+=1
            idx2+=1

Function Sort_pancake(P,a,buffer,front,end):
    if end > front :
        mid = (front+end)/2
```

```
        Sort_pancake(P,a,buffer,front,mid)
        Sort_pancake(P,a,buffer,mid+1,end)
        Merge(P,a,buffer,front,mid,end)
```

Subproblem 3:

利用二分搜，把目標、最後一個、二分搜抓出來的拿去問鬆餅神，如果吐出來的是目標則往尾端繼續二分搜，否則往頭二分搜直到搜到重覆的點

```
if m== 1 :
    insert target to 2
    return
if m==2:
    if Pancake-God-Oracle(L, 0,target,1) == 0:
        insert target to 0
    if Pancake-God-Oracle(L, 0,target,1) == 1:
        insert target to 2
    if Pancake-God-Oracle(L, 0,target,1) == target:
        insert target to 1
    return

mid = end/2
boundary = end
if Pancake-God-Oracle(L, mid,target,end) == mid:
    boundary = front
Function Bisearch(P, mid, boundary, path, target):
    new_mid = (boundary+mid)/2
    if mid == front:
        insert target to mid+1
        return
    if Pancake-God-Oracle(L, new_mid, target, boundary)==target:
        Bisearch(P,new_mid,boundary,path,target)
    else:
        Bisearch(P,mid,new_mid,path,target)
```

Subproblem 4:

如果數量>1，就先塞兩個元素進list
接著利用subproblem 3 二分搜的方式把剩下的元素插到正確的地方
每次都$log(n)$共$n-2$個元素，共$nlogn$

Subproblem 5:

ref:和李長諺、陳柏諺討論
將一個大小為n的陣列依任意方式作排列後會有$n!$種情況，而排列正確的只有升冪和降冪兩種。而每次$query$ 後都會將錯誤排列的可能性砍掉。
因此可將他視為一棵二元樹，向左為降冪向右為升冪。而正確排列的葉子只有最左和最右兩片葉子。而樹上需有$n!$片葉子。二元樹最底層會有2^k片葉子(k是樹的高度)，每次搜尋的時間複雜度也為樹的深度。
因此$2^k \geq n! \quad \therefore k \geq lg(n!)$
根據小O定義，可以得知只要證明k為$\Omega(nlog(n)$即可k不是$o(nlog(n))$
由**HW1 P1-8**可以得知:
$lg(n!) \geq \frac{n}{2}lg(\frac{n}{2}) = \frac{nlg(n)}{2} - \frac{n}{2}$

$lg(n!) = \Omega(nlog(n))$
因此$k = \Omega(nlog(n))$,故不為$o(nlog(n))$

Subproblem 6:

每次搜到else if的時候會對t_r, t_l做排列(排成遞減)再一路回傳到最上層，可當作是先排列前$\frac{2}{3}n$項，再排列後$\frac{2}{3}n$項，最後再排列前$\frac{2}{3}n$項。每次排列完後，後半段的項一定小於前半段的項，因此分別排列前後前$\frac{2}{3}n$項可以使得應該放在前$\frac{2}{3}n$區域卻位在後$\frac{2}{3}n$的項被搬到前面來，反之亦同。因此這個演算法可以將數列照遞減做排序。

Subproblem 7:

Ans : $T(n) = 3T(\frac{2}{3}n) + \Theta(1)$
每次呼叫都會再往下呼叫三次ELF_Sort，並且把範圍縮到原本的$\frac{2}{3}$。因每次算△都是constant time所以$f(n) = 1$
因此$a = 3\ b = \frac{2}{3}$

Subproblem 8:

$T(n) = 3T(\frac{2}{3}n) + \Theta(1)$
$T(\frac{2}{3}n) = 3T((\frac{2}{3})^2n) + \Theta(1)$
.
.
.
$T(\frac{3}{2}) = 3T(1) + \Theta(1)$
共$|log_{\frac{2}{3}}n| = log_{1.5}n$項，移項可得:
$T(n) + \frac{1}{2}\Theta(1) = 3(\ T(\frac{2}{3}n) + +\frac{1}{2}\Theta(1)\ )$
$T(\frac{2}{3}n) + \frac{1}{2}\Theta(1) = 3(\ T((\frac{2}{3})^2n) + \frac{1}{2}\Theta(1)\ )$
.
.
.
$T(2) + \frac{1}{2}\Theta(1) = 3(\ T(1) + \frac{1}{2}\Theta(1)\ )$
相乘得到:
$T(n) + \frac{1}{2}\Theta(1) = 3^{log_{1.5}n}(\ T(1) + \frac{1}{2}\Theta(1)\ )$
$= n^{log_{1.5}3}(\ T(1) + \frac{1}{2}\Theta(1)\ )$
合併後可以得到$T(n) = \Theta(n^{log_{1.5}3})$
因此for all n>1 there must be a $c = 1$ that $n^{log_{1.5}3} = T(n) \leq cn^3$
$T(n) = O(n^3)$

Problem 2

Subproblem 1:

為了找到比t_k小而最接近它的數，它必在t_k左子樹的最右邊，或是它的父節點以及他的祖先

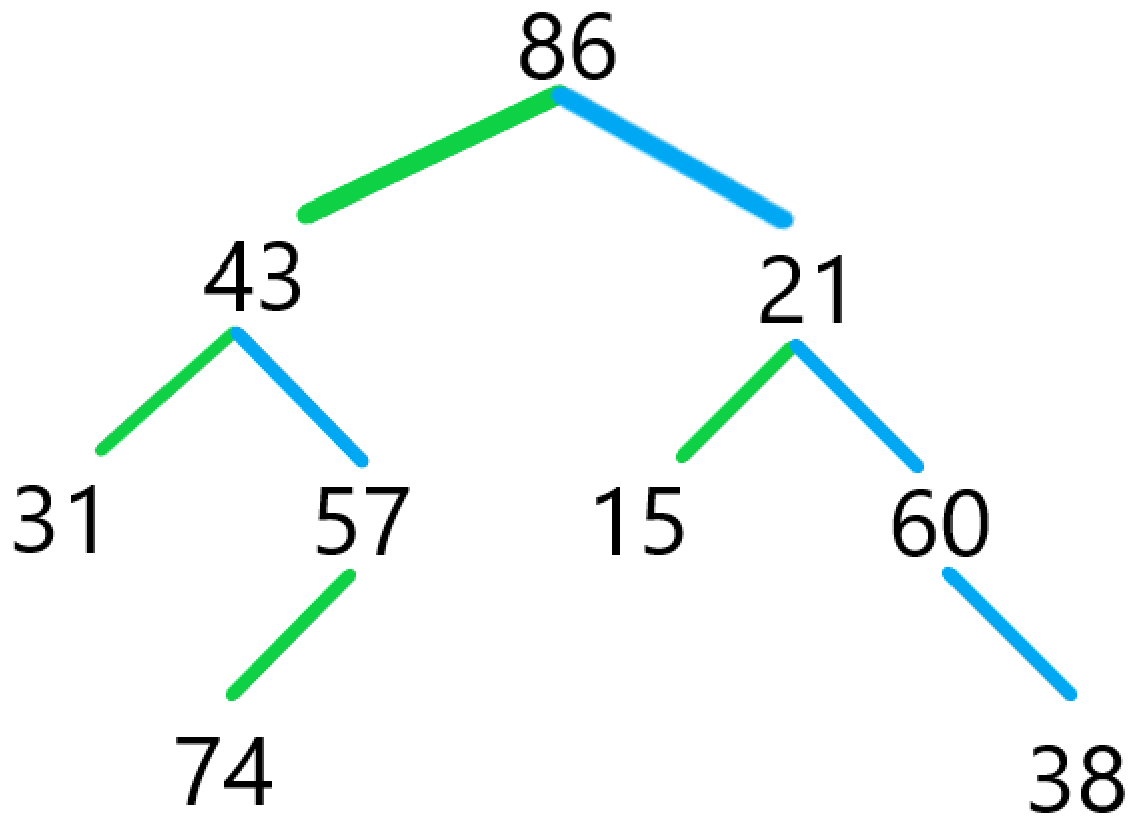
```
node *now = tk->left
node *parent = tk->parent
if now!=NULL:
    while now->right != NULL:
        now = now->right
    t(k-1) = now
else:
    while parent->parent!=NULL or parent->parent->left == parent:
        parent = parent->parent
    t(k-1) = parent
```

Subproblem 2:

因為左子樹必小於父節點，而整棵樹中最右方的葉子必最大，因此佐子樹中最右邊的葉子必為小於t_k又最接近t_k的節點。而若t_k位在父節點的右邊且左子樹為空時，還要不斷向上確認父節點和祖先直到當前確認的node不在他的父節點的右子樹，因為只有如此父節點才會小於當前節點。

Subproblem 3:

如圖:



Subproblem 4:

Ans = 否
當樹的大小為1時，明顯地相同的inoder和preorder會是相同的tree
而每棵大小為n的樹可以由根分為左子樹、根、右子樹。假設根在inorder中的位置為rt_1則左子樹為[1, rt_1] 右子樹為[rt_1, n]。
preorder中的位置為rt_2則左子樹為[rt_1, rt_1+len(Ltree)] 右子樹為[rt_1+len(Ltree), n]。
假設T_1和T_2有相同$inorder, preorder$，則T_1和T_2有在inorder中有相同的左子樹和右子樹、$preorder$中也是。
當遞迴到最底時可以知道他們的葉子相同，因此兩棵樹的形狀必定相同。

Subproblem 5:

先利用preorder找出每個node的data，再到inorder裡找出那點，它左邊的data為左子樹，右邊的為右子樹。再一路遞回下去下去

```
node *construct_tree(int preorder[],int subs[],int idx)
    node *now
    now->data = preorder[idx]
    if lens(subs) == 1:
        now->left = NULL
        now->right = NULL
        return now
    int left[],right[],a=0
    for a from 0 to lens(subs)
        if subs[a] = preorder[idx]:
            break

    now->left = *construct_tree(preorder[],subs[0, a-1],idx+1)
    now->right = *construct_tree(preorder[],subs[a+1, lens(subs)],idx+a)
    return now
node *Head = construct_tree(preorder[], inorder, 0)
```

Problem 3

Subproblem 1:

把value改完後確認是否符合min-heap

```
Function h_modify(x, v):
    h[x]->value = v
    parent = h[x]->parent
    if(parent->value > h[x]->value)
        while(parent->value > h[x]->value):
            swap(parent->value , h[x]->value)
            h[x] = parent
            if parent->parent ==NULL:
                break
            else:
                parent = parent->parent
    else if h[x]->value > h[x]->left->value or h[x]->value > h[x]->right->value
        while h[x]->value > h[x]->left->value or h[x]->value > h[x]->right-
>value
            if min(h[x]->right->value,h[x]->left->value)==h[x]->right->value:
                swap(h[x], h[x]->right)
            else
                swap(h[x], h[x]->left)
```

改完後要向上或向下確認是否符合$min-heap$
最糟情況時要從葉子走到樹根，或是從樹根走到葉子。
如此需耗費h的時間(h為樹高)，而$h = lg(h)$
因此時間複雜度為$O(lg(h))$

Subproblem 2:

E=Empty

(a)

E	E	E	E
E	E	E	E
E	E	E	1
4	E	E	2

(b)

E	E	E	E
E	E	E	E
E	E	E	1
4	E	E	E

(c)

E	E	E	E
E	E	E	3
E	E	E	1
4	E	E	E

(d)

E	E	E	E
E	E	E	3
E	E	E	E
4	E	E	E

(e)

E	E	E	E
E	E	E	E
E	E	E	E
4	E	E	E

Subproblem 3:

先對每行每列建立線段樹維護區間最小值
每棵樹的head是全部範圍的最小值，左子樹紀錄每個父節點的區間最小到區間中點區間的最小值，右子樹紀錄區間中點到父節點的區間最大區間的最小值
Ex 父節點紀錄 $1\ to\ n$ 、左子節點紀錄$1\ to\ \frac{n+1}{2}$區間的最小值、右子節點紀錄$\frac{n+1}{2}+1\ to\ n$區間的最小值

```
struct seg-tree{
    int min
    int head
    int end
    int row
    int col
    struct seg-tree left
    struct seg-tree right
    struct seg-tree parent
}
```

Subproblem 4:

D.add(i, j, v):
分別在紀錄行和列的陣列裡找出第i行和第j列的線段樹 $Tree_i, Treej$
接著再將v插入這兩個線段樹中，在$Tree_i$中重新比對和位置j有關的最小值是否改變。最多要比對h層，$h = lg(n)$
同理也需要對$Treej$做一樣的事，只是換成比對和位置i相關的最小值是否改變
因此時間複雜度為$lg(n) + lg(m) = lg(nm)$

```
//更新tree_i
while tree_i->left!=NILL or tree_i->right!=NILL:
    int mid = (tree_i->head+tree_i->end)/2
    if j>mid
        tree_i = tree_i->right
    else tree_i = tree_i->left
tree_i->min = v
while tree_i->parent != NILL:
    if tree_i->min < tree_i->parent->min:
        tree_i->parent->min = tree_i->min
    tree_i = tree_i->parent
//更新tree_j
while tree_j->left!=NILL or tree_j->right!=NILL:
    int mid = (tree_j->head+tree_j->end)/2
    if i>mid
        tree = tree->right
    else tree = tree->left
tree_j->min = v
```

```
while tree_j->parent != NILL:
    if tree_j->min < tree_j->parent->min:
        tree_j->parent->min = tree_j->min
    tree_j = tree_j->parent
```

D.extractMinRow(i):

先向下走到葉子將最小值更新成空(正無限)，並記錄其位置(i, j)
接著開始和其父節點比較，最後一層一層回推到最上面的節點
這樣要向下走到底再走到頂一次，需要$O(2lg(n)) = O(lg(n))$
接著找出第j列的線段樹並一路走到j的葉子，更新成空後再一路走回最上面

```
while tree_i->left!=NILL or tree_i->right!=NILL:
    int mid = (tree_i->head + tree_i->end)/2
    if j>mid
        tree_i = tree_i->right
    else tree_i = tree_i->left

old = tree_i->min
tree_i->min = inf
index = tree_i->col

while tree_i->parent != NILL:
    if tree_i->min == old and tree_i->left->min == inf:
        tree_i->min = tree_i->right->min
    else if tree_i->min == old and tree_i->right->min == inf:
        tree_i->min = tree_i->left->min
    tree_i = tree_i->parent
//更新tree_j
while tree_j->left!=NILL or tree_j->right!=NILL:
    int mid = (tree_j->head+tree_j->end)/2
    if i>mid
        tree = tree->right
    else tree = tree->left
tree_j->min = inf
while tree_j->parent != NILL:
    if tree_-j>min == old and tree_j->left->min == inf:
        tree_j->min = tree_j->right->min
    else if tree_j->min == old and tree_j->right->min == inf:
        tree_j->min = tree_j->left->min
    tree_j = tree_j->parent
```

D.extractMinCol(j):

方法同**D.extractMinRow(i):**，只是需將被刪除元素的行列對調

```
while tree_j->left!=NILL or tree_j->right!=NILL:
    int mid = (tree_j->head + tree_j->end)/2
    if j>mid
        tree_j = tree_j->right
    else tree_j = tree_j->left

old = tree_j->min
tree_j->min = inf
index = tree_j->col

while tree_j->parent != NILL:
    if tree_j->min == old and tree_j->left->min == inf:
```

```
        tree_j->min = tree_j->right->min
    else if tree_j->min == old and tree_j->right->min == inf:
        tree_j->min = tree_j->left->min
    tree_j = tree_j->parent
//更新tree_i
while tree_i->left!=NILL or tree_i->right!=NILL:
    int mid = (tree_i->head+tree_i->end)/2
    if i>mid
        tree_i = tree_i->right
    else tree_i = tree_i->left
tree_i->min = inf
while tree_i->parent != NILL:
    if tree_-i>min == old and tree_i->left->min == inf:
        tree_i->min = tree_i->right->min
    else if tree_i->min == old and tree_i->right->min == inf:
        tree_i->min = tree_i->left->min
    tree_i = tree_i->parent
```

D.delete(i, j):

先叫出第i行和第j列的線段樹

接著一路向下走到i, j的葉子，修改成空後一路向上更新

需要從頭走到最下面再走回到最上面，因此時間複雜度為$lg(n) + lg(m) = lg(nm)$

```
while tree_j->left!=NILL or tree_j->right!=NILL:
    int mid = (tree_j->head + tree_j->end)/2
    if j>mid
        tree_j = tree_j->right
    else tree_j = tree_j->left

old = tree_j->min
tree_j->min = inf
index = tree_j->col

while tree_j->parent != NILL:
    if tree_j->min == old and tree_j->left->min == inf:
        tree_j->min = tree_j->right->min
    else if tree_j->min == old and tree_j->right->min == inf:
        tree_j->min = tree_j->left->min
    tree_j = tree_j->parent
//更新tree_i
while tree_i->left!=NILL or tree_i->right!=NILL:
    int mid = (tree_i->head+tree_i->end)/2
    if i>mid
        tree_i = tree_i->right
    else tree_i = tree_i->left
tree_i->min = inf
while tree_i->parent != NILL:
    if tree_-i>min == old and tree_i->left->min == inf:
        tree_i->min = tree_i->right->min
    else if tree_i->min == old and tree_i->right->min == inf:
        tree_i->min = tree_i->left->min
    tree_i = tree_i->parent
```