Problem 1

1.

```
function GetBoundary(P)
    bound1 = 1
    bound2 = 2
    n = P.len
    for i from 3 to n:
        query_result = PancakeGodOracle(P, bound1, bound2, i)
        if query_result == i:
            continue
        else if query_result == bound1:
            bound1 = i
        else if query_result == bound2:
            bound2 = i
    return bound1, bound2
```

Query is done for `n-2` times, therefore the query complexity is $O(n)$

2.

```
/*
A[:-1] means the whole array A except the last element
A[2:] means the whole array A element except the first element
*/
function SortPancakes(P)
    n = P.len
    left_bound, right_bound = GetBoundary(P)
    swap(P, 1, left_bound); swap(P, n, right_bound)
    function QSort(P)
        m = P.len
        if m <= 2:
            return P
        else:
            pivot = random(2, m-1)
            l_pt = array(P[1]); r_pt = array(P[pivot])
            for i from 2 to m-1 except pivot:
                query = PancakeGodOracle(P, 1, pivot, i)
                if query == i:
                    l_pt.append(P[i])
                else:
                    r_pt.append(P[i])
```

```
            l_pt.append(P[pivot]); r_pt.append(P[m])
            /* pivot will be in both l_pt and r_pt, so remove it when returning
    value */
            return QSort(l_pt)[:-1] + P[pivot] + QSort(r_pt)[2:]
    P = QSort(P)
```

My implementation is a pancake variation of quicksort. The recursion depth on average would be $O(\log n)$, and in each depth the query complexity is $O(n)$, therefore the total query complexity is $O(n \log n)$

3.

```
function InsertPancake(L, new_pancake)
    l = 1; r = L.len
    L.append(new_pancake)
    new_p = L.len
    final_pos = -1
    while l < r:
        mid = floor((l+r)/2)
        query = PancakeGodOracle(P, l, mid, new_p)
        if query == new_p:
            r = mid-1
        else:
            l = mid+1
    final_pos = l
    for i from final_pos to L.len-1
        swap(L, i, L.len)
```

My implementation is a pancake variation of binary search. The query complexity can be easily found to be $O(\log n)$

4.

```
/* A[1:i+1] means the array A from first element to i-th element */
function SortPancakesAgain(P)
    for i from 2 to P.len
        InsertPancake(P[1:i+1], P[i])
```

`InsertPancake()` runs `n` times, therefore the query complexity is $O(n \log n)$

5.

Skipped

6.

For $n = 1$, `P` is already in descending order and `ELF-SORT()` does nothing to `P`.

For $n = 2$, if `P[2] > P[1]`, they will be swapped and then `P` is in descending order.

For $n = 3$, the code would look like:

```
ELF-SORT(P, 1, 2)
ELF-SORT(P, 2, 3)
ELF-SORT(P, 1, 2)
```

The first two line moves the smallest element in `P` to the end, and the last line sorts `P[1]` and `P[2]`. `P` will be in descending order.

Assume that $\forall\, n \leq k$, `ELF-SORT(P, 1, n)` sorts `P[1]` to `P[n]` in descending order. As shown above, this is true $\forall\, n \leq 3$.

`ELF-SORT(P, 1+t, n+t)` also sorts `P[1+t]` to `P[n+t]` in descending order because it's just a shift on `P`.

For $n = k + 1$, the code would look like:

```
Delta = floor((k+1)/3)
ELF-SORT(P, 1, k+1 - Delta)
ELF-SORT(P, 1 + Delta, k+1)
ELF-SORT(P, 1, k+1 - Delta)
```

Because we are only considering $n \geq 4$ in this part, `Delta` or Δ is always larger then 1. Therefore all those three `ELF-SORT()` would sort the respective range in descending order.

The first two `ELF-SORT()` will sort the least $\frac{n}{3}$ element to the correct place, and the last one will sort the first $\frac{2n}{3}$ elements to the correct place. Therefore `P` is sorted in descending order.

Since `ELF-SORT(P, 1, n)` works for $n = 1, 2, 3$, and if for $n = k$ `ELF-SORT` works, it would also work for $n = k + 1$. By mathematical induction, `ELF-SORT(P, 1, n)` sorts `P` in descending order $\forall\, n \in \mathbb{N}$.

7.

For $n \geq 3$, the code would look like:

```
Delta = floor(n/3)
ELF-SORT(P, 1, n - Delta)
ELF-SORT(P, 1 + Delta, n)
ELF-SORT(P, 1, n - Delta)
```

From this, it's obvious that $T(n) = 3T(\frac{2}{3}n) + \Theta(1)$, because `floor()` runs at constant time

For $n = 2$, the running time is the time of two comparisons and a swap, which is $\Theta(1)$.

For $n = 1$, the running time is the time of two comparisons, which is $\Theta(1)$.

8.

By the recurrence relation we have:

$$\begin{aligned}
T(n) &= 3^1 T((\frac{2}{3})^1 n) + \Theta(1) \\
3^1 T((\frac{2}{3})^1 n) &= 3^2 T((\frac{2}{3})^2 n) + 3^1 \Theta(1) \\
3^2 T((\frac{2}{3})^2 n) &= 3^3 T((\frac{2}{3})^3 n) + 3^2 \Theta(1) \\
&\dots \\
3^{k-1} T((\frac{2}{3})^{k-1} n) &= 3^k T(1) + 3^{k-1}\Theta(1) \ \lor \ 3^k T(2) + 3^{k-1}\Theta(1)
\end{aligned}$$

Summing all k equations and replacing $T(1)$ and $T(2)$ with $\Theta(1)$ yields:

$$\begin{aligned}
T(n) &= 3^k \Theta(1) + \sum_{i=0}^{k-1} 3^i \cdot \Theta(1) \\
&= \sum_{i=0}^{k} 3^i \cdot \Theta(1) \\
&= \frac{3^{k+1} - 1}{3 - 1} \\
&= \frac{3}{2} 3^k - \frac{1}{2}
\end{aligned}$$

And because $k = \lfloor \log_{\frac{3}{2}} n \rfloor$, we have:

$$
\begin{aligned}
T(n) &= \frac{3}{2}3^k - \frac{1}{2} \\
&= \frac{3}{2}3^{\lfloor \log_{1.5} n \rfloor} - \frac{1}{2} \\
&\leq \frac{3}{2}3^{1+\log_{1.5} n} - \frac{1}{2} \\
\frac{3}{2}3^{1+\log_{1.5} n} - \frac{1}{2} &= \frac{9}{2}n^{\log_{1.5} 3} - \frac{1}{2} \\
&\leq \frac{9}{2}n^3
\end{aligned}
$$

Choose $n_0 = 1$ and $c = \frac{9}{2}$, we have:

$$
\begin{aligned}
&\forall\ n \geq n_0 = 1, \\
T(n) &\leq \frac{3}{2}3^{1+\log_{1.5} n} - \frac{1}{2} \\
&\leq \frac{9}{2}n^3 \\
&= c \cdot n^3 \\
\Rightarrow T(n) &= O(n^3)
\end{aligned}
$$

Problem 2

1.

```
function FindPrev(T, t_k)
    node = T.root
    prev_node = NIL
    while node != NIL
      if node.key >= t_k
          node = node.left
      else
          prev_node = node
          node = node.right
    return prev_node
```

2.

If the key larger or equal to t_k, t_{k-1} must be in the left subtree, therefore we go left.

If the key is smaller than t_k, there are two possibilities:

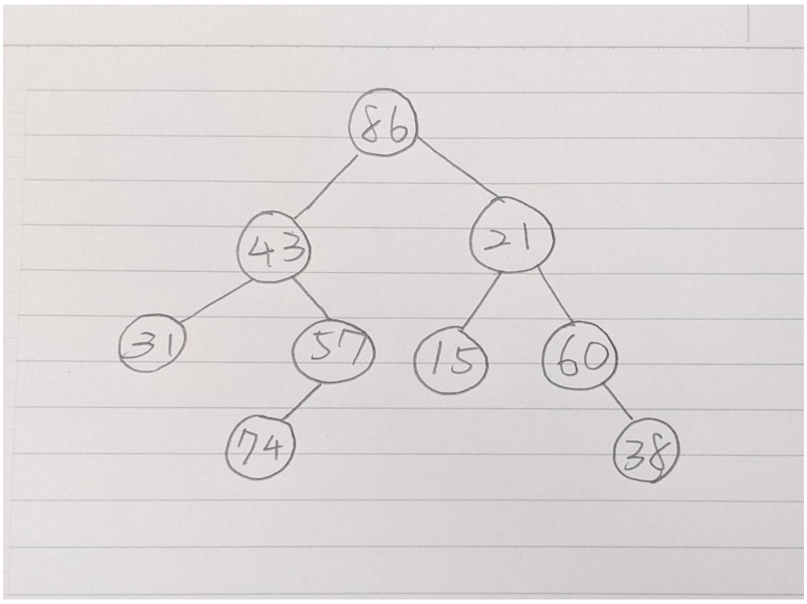
1. It's t_{k-1}
2. It is smaller than t_{k-1}

For case 1, because we go to the right subtree, every key is larger than t_{k-1} and larger or equal to t_k. `prev_node = node` won't be executed anymore and the algorithm works.

For case 2, t_{k-1} must be in the right subtree, therefore we go right.

If it receives t_1 as input, we always go left and `prev_node = node` won't be executed. The return value would be `NIL`.

3.



4.

Because `preorder[1]` is the root of the tree, by locating it in `inorder` we can extract (`inorder` , `preorder`) pair of left subtree and right subtree. Therefore if two trees have the same (`inorder` , `preorder`) pair, the root must be the same, two left subtrees have the same (`inorder` , `preorder`) pair, and two right subtrees also have the same (`inorder` , `preorder`) pair.

Doing this recursively for subtrees can show that this two trees must be the same.

5.

```
/* A.index(v) returns the index of v in array A */
function Reconstruct(inorder, preorder)
    if inorder.len == 0:
        return NIL
    rt = root()
    rt.key = preorder[1]
    l_size = inorder.index(rt.key) - 1
    r_size = inorder.len - l_size - 1
    l_inorder = inorder[:l_size+1]
    l_preorder = preorder[2:l_size+2]
    r_inorder = inorder[l_size+2:]
    r_preorder = preorder[l_size+2:]
    rt.left = Reconstruct(l_inorder, l_preorder)
    rt.right = Reconstruct(r_inorder, r_preorder)
    return rt
```

Time complexity: $O(n^2)$

Problem 3

1.

```
function modify(x, v)
    if (v > x.key)
        x.key = v
        min_child = minNode(x.l, x.r)
        while (min_child != NIL)
            if x.key <= min_child.key
                break
            else
                swapNode(x, min_child)
                min_child = minNode(x.l, x.r)
    else if (v < x.key)
        x.key = v
        parent = x.p
        while (parent != NIL)
            if x.key >= parent.key
                break
            else
                swapNode(x, parent)
                parent = x.p
```

When `v > x.key`, in the `while` loop we always try to move `x` down, and a node can be moved down for at most $\lg |h|$ times (from top to bottom). `swapNode` and `minNode` can both be done in $O(1)$ time. Therefore time complexity is $\lg |h| \cdot O(1) = O(\lg |h|)$.

When `v < x.key`, instead of moving down, we are trying to move `x` up. The same analogy from above applies, therefore time complexity is also $O(\lg |h|)$.

2.

Empty locations are left blank

(a)

$$A_{4\times 4} = \begin{bmatrix} & & & \\ & & & \\ & & & 1 \\ 4 & & & 2 \end{bmatrix}$$

(b)

$$A_{4\times 4} = \begin{bmatrix} & & & \\ & & & \\ & & & 1 \\ 4 & & & \end{bmatrix}$$

(c)

$$A_{4\times 4} = \begin{bmatrix} & & & \\ & & & 3 \\ & & & 1 \\ 4 & & & \end{bmatrix}$$

(d)

$$A_{4\times 4} = \begin{bmatrix} & & & \\ & & & 3 \\ & & & \\ 4 & & & \end{bmatrix}$$

(e)

$$A_{4\times 4} = \begin{bmatrix} & & & \\ & & & \\ & & & \\ 4 & & & \end{bmatrix}$$

3.

Elements are stored in a $N \times M$ array `A`. Each row and column has a corresponding heap `row[i]` / `col[j]`. Each elements has these extra attributes: `col_l`, `col_r`, `row_l`, `row_r`, representing its left/right child in the row/column heap. Each element also has `i`, `j` representing its index.

4.

```
/* Using heap operations from P3-1 */
/* Assmuing that "extract" operations return the extracted element */
function add(i, j, v)
    A[i][j] = v
    row[i].insert(A[i][j])
    col[j].insert(A[i][j])
```

```
function extractMinRow(i)
    row_min = row[i].extractMin()
    col[row_min.j].delete(row_min)

function extractMinCol(j)
    col_min = col[j].extractMin()
    row[col_min.i].delete(col_min)

function delete(i, j)
    row[i].delete(A[i][j])
    col[i].delete(A[i][j])
```

Maximum size of heap `row[i]` is the number of columns M. And the maximum size of heap `col[j]` is the number of rows N.

`add()`

`A[i][j] = v` is $O(1)$.
`row[i].insert()` is $O(\lg M)$. `col[j].insert()` is $O(\lg N)$. (heap operation)
Total time complexity is $O(1) + O(\lg M) + O(\lg N) = O(\lg(MN))$.

`extractMinRow()`

`row_min = row[i].extractMin()` is $O(\lg M)$. (heap operation)
`col[row_min.j].delete(row_min)` is $O(\lg N)$. (also heap operation)
Total time complexity is $O(\lg M) + O(\lg N) = O(\lg(MN))$.

`extractMinCol()`

`col_min = col[j].extractMin()` is $O(\lg N)$. (heap operation)
`row[col_min.i].delete(col_min)` is $O(\lg M)$. (also heap operation)
Total time complexity is $O(\lg N) + O(\lg M) = O(\lg(MN))$.

`delete()`

`row[i].delete(A[i][j])` is $O(\lg M)$. (heap operation)
`col[i].delete(A[i][j])` is $O(\lg N)$. (also heap operation)
Total time complexity is $O(\lg M) + O(\lg N) = O(\lg(MN))$.