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:

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

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:

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

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

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