P3

References:

None

1.

Algorithm

Workflow:

1. Create an array `visited` with the same length as `A`, and set all values to `False`. It would keep track of if the position on `A` has been visited.
2. Set `cur` equal to the initial position.
3. Repeat the following things until return:
    1. If `cur` is the same as our next position (which is `A[cur]`), then return "will stop".
    2. If `visited[cur]` is `True`, it means we are in a loop, return "won't stop".
    3. Else, we set `visited[cur]` to `True`, and set `cur` to the next position.

Written in pseudo code:

```
func judgeStop(A, start):
    A_len = A.len()
    visited[A_len] = {False}
    cur = start
    while(cur != A[cur]):
        if(visited[cur] = True):
            return False
        else:
            visited[cur] = True
            cur = A[cur]
    return True
```

We know that the frog will either stop at some point or go into a loop. When the frog will stop, the algorithm obviously works. When the frog will go in to a loop, since the array has a finite size, the loop also has a finite size, and that means the frog will visit a position twice. Therefore the algorithm will also work in this scenario.

Time Complexity & Extra-space Complexity

In the worst case, my algorithm will traverse the entire array `A` then stop, therefore the time complexity would be $O(n)$.

For extra-space complexity, the additional variables I used are `A_len`, `visited`, and `cur`, and they respectively take up $O(1)$, $O(n)$, and $O(1)$ spaces. Therefore the extra-space complexity in total is $O(n)$.

2.

Algorithm

Workflow:

1. Create an array `visited` with the same length as `A`, and set all values to `0`. It would keep track of at which iteration is the position visited.
2. Set `cur` equal to the initial position. Set `cnt = 1`, which is the counter of iteration times.
3. Repeat the following things until return:
    1. If `visited[cur]` isn't `0`, it means we have completed a loop. Therefore we return `cur - visited[cur]`, which is the length of the loop.
    2. Else, we set `visited[cur]` to `cnt`, `cur` to the next position, and add `1` to `cnt`.

Written in pseudo code:

```
func getLoopLen(A, start):
    A_len = A.len()
    visited[A_len] = {0}
    cur = start
    cnt = 1
    while(True):
        if(visited[cur] != 0):
            return cnt - visited[cur]
        else:
            visited[cur] = cnt
            cur = A[cur]
            cnt = cnt+1
```

Time Complexity & Extra-space Complexity

Because there is only one position we would visit twice, the worst time complexity possible would be $O(n)$ (when the loop is as large as the whole array).

For extra-space complexity, the additional variables I used are `A_len`, `visited`, `cur`, and `cnt`, and they respectively take up $O(1)$, $O(n)$, $O(1)$, and $O(1)$ spaces. Therefore the extra-space complexity in total is $O(n)$.

3.

Algorithm

Math stuff:

A is a stricly increasing array

$\Rightarrow \forall\, m > n, a_m > a_n$

By median's property, we have:

$a_0 \leq M_{0,i} \leq a_{i-1},\ a_i \leq M_{i,j} \leq a_{j-1},\ a_j \leq M_{j,n} \leq a_{n-1}$

$\Rightarrow M_{0,i} < M_{i,j} < M_{j,n}$

$\Rightarrow f(i,j) = M_{j,n} - M_{0,i}$

To minimize $f(i,j)$, we need $j = i + 1$ because:

$\forall\, j > i+1,\ a_{(j+n-1)/2} > a_{(i+1+n-1)/2}$

$\Rightarrow \forall\, j > i+1,\ M_{j,n} > M_{i+1,n}$

$\Rightarrow \forall\, j > i+1,\ f(i,j) > f(i,i+1)$

Workflow:

1. Initiallize `current_min`, `min_i`, and `min_j`
2. Iterate `i` from `1` to `n-2`
3. In each interation, let `j=i+1`, calculate median of `A[j:n]` and `A[0:i]`, then substract them to get $f(i,j)$
4. Update `current_min`, `min_i`, and `min_j` if the current $f(i,j)$ is smaller
5. Return `min_i` and `min_j` when the loop ends

Written in pseudo code:

```
minimizeF(A, n)
    current_min = INF
    min_i, min_j = -1, -1
    for i from 1 to n-2
        j = i+1
        f = median(A[j:n]) - median(A[0:i])
        if f < current_min
            current_min = f
            min_i = i
            min_j = j
    return min_i, min_j
```

Time Complexity & Extra-space Complexity

Getting the median of an array is only $O(1)$ because the index can be calculated given the start and end index. Plus my algorithm runs a single for loop, therefore the time complexity would be $O(n)$.

All extra variables have constant space despite n, therefore the extra-space complexity is $O(1)$.

4.

Algorithm

Workflow:

1. Traverse the circular linked list and find the two decreasing node, name them `h1` and `h2`.
2. Save a copy of `h1` and `h2` as `end2` and `end1`.
3. Set `new_head` point to the smaller one between `h1` and `h2`, and let the chosen node go to next node.

4. Set `cur_node=new_head`. Treat `h1` and `h2` as if they are two linked lists, merge them into `new_head`. Merging is to choose the smaller one between two nodes, link `cur_tail` to it, then let the chosen node go to next node.
5. When one of `h1` and `h2` has gone to the end, go to the end of the leftover linked list, update the tail's `next`, then connect the whole list to `cur_tail`.

Written in pseudo code:

```
sortL(head)
    // get the two decreasing node
    h1, h2 = NIL, NIL
    cur_node = head
    while h2 == NIL
        if cur_node.value > cur_node.next.value
            if h1 == NIL
                h1 = cur_node
            else
                h2 = cur_node
        cur_node = cur_node.next
    end1, end2 = h2, h1

    // assign value to new_head
    new_head = NIL
    if h1.value < h2.value
        new_head = h1
        h1 = h1.next
    else
        new_head = h2
        h2 = h2.next

    // merge
    cur_node = new_head
    while True
        if h2 == end2
            // swap h1,h2 and end1,end2 for cleaner code
            h1, h2 = h2, h1
            end1, end2 = end2, end1
        if h1 == end1
            cur_node.next = h2
            while h2.next != end2 // go to the tail node
                h2 = h2.next
            h2.next = new_head
            break
        else
            if h1.value < h2.value
                cur_node.next = h1
                h1 = h1.next
            else
                cur_node.next = h2
                h2 = h2.next
            cur_node = cur_node.next
```

Time Complexity & Extra-space Complexity

Time complexity of each step in workflow are: 1. $O(n)$ 2. $O(1)$ 3. $O(1)$ 4. $O(n)$ 5. $O(n)$, therefore the total time complexity is $O(n)$

All extra variables used have constant space despite n, therefore the total extra-space complexity is $O(1)$.