Problem 1

References:

None

1.

In the k-th iteration of the while loop, $sum = 1 + 2 + \cdots + k = \frac{k(k+1)}{2}$

$\Rightarrow$ total iteration time x satisfies $\frac{x(x-1)}{2} < n \leq \frac{x(x+1)}{2}$ $\Rightarrow$ time complexity $x = \Theta(\sqrt{n})$

2.

In the k-th iteration, $m = 2^{2^{k-1}}$

$\Rightarrow$ total iteration time x satisfies $2^{2^{x-2}} < n \leq 2^{2^{x-1}}$ $\Rightarrow$ time complexity $x = \Theta(\sqrt{n})$

3.

For $n > 87506055$, total operation $x = 1 + 4 + \cdots + 4^{n-k} + 4^{n-k} \cdot 3 + \cdots + 4^{n-k} \cdot 3^k$, where $k = 87506055$

$\Rightarrow$ time complexity $x = \Theta(4^n)$

4.

$\because f(n), g(n)$ are both positive $\therefore max(f(n),\ g(n)) \leq f(n) + g(n) \leq 2 \cdot max(f(n),\ g(n))$

$\Rightarrow f(n) + g(n) = \Theta(max(f(n),\ g(n)))$

5.

$f(n) = O(i(n)) \Rightarrow \exists\ c_1 > 0,\ n_1 > 0\ s.t.\ \forall\ n > n_1,\ f(n) \leq c_1 \cdot i(n)$

$g(n) = O(j(n)) \Rightarrow \exists\ c_2 > 0,\ n_2 > 0\ s.t.\ \forall\ n > n_2,\ g(n) \leq c_2 \cdot j(n)$

Let $n' = max(n_1,\ n_2),\ c' = c_1 \cdot c_2$, multiplying the first two lines we have

$\forall\ n > n',\ f(n) \cdot g(n) \leq c' \cdot i(n) \cdot j(n) \Rightarrow f(n) \cdot g(n) = O(i(n) \cdot j(n))$

6.

Choose $f(n) = lg3 \cdot n$, $g(n) = n$, then $f(n) = O(g(n))$, and $2^{f(n)} = 2^{lg3 \cdot n} = 3^n$, $2^{g(n)} = 2^n$

Assume that $3^n = O(2^n) \Rightarrow \exists$ finite n_0, c such that $\forall\ n > n_0,\ 3^n \leq c \cdot 2^n \Rightarrow \forall\ n > n_0,\ (\frac{3}{2})^n \leq c$

But $\lim_{n\to\infty}(\frac{3}{2})^n = \infty \Rightarrow c \geq \infty \Rightarrow$ assumption is false, $3^n \neq O(2^n)$

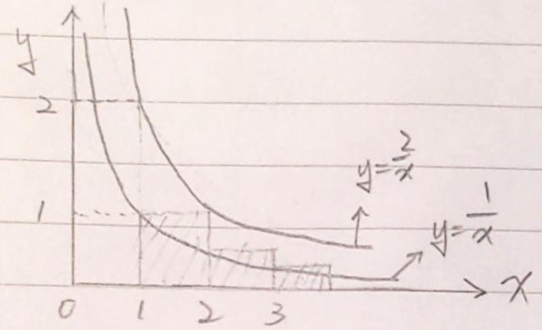
$\Rightarrow 2^{f(n)} \neq O(2^{g(n)})$

7.

$$\sum_{k=1}^{n}\frac{1}{k} = \sum_{k=1}^{n} 1\cdot\frac{1}{k} \ge \int_{1}^{n+1}\frac{1}{x}dx = \ln(n+1) > \ln(n) = \frac{\lg n}{\lg e} = \ln 2\cdot\lg n \quad \forall n>1$$

— ①

$$\frac{\frac{1}{k}}{\frac{1}{k+1}} = \frac{k+1}{k} = 1+\frac{1}{k} \le 2 \ \forall k\ge 1 \Rightarrow \frac{1}{k+1}\cdot 2 \ge \frac{1}{k} \ \forall k\ge 1$$



$$\Rightarrow \sum_{k=1}^{n}\frac{1}{k} \le \int_{1}^{n+1}\frac{2}{x}dx = 2\ln(n+1)$$

$$2\ln(n+1) = 2\cdot\frac{\ln(n+1)}{\ln(n)}\cdot\ln(n)$$

$$\le 2\cdot\frac{\ln 3}{\ln 2}\cdot\ln 2\cdot\lg n \quad \forall n\ge 2$$

— ②

①: $\sum_{k=1}^{n}\frac{1}{k} \ge \ln 2\cdot\lg n \underset{\forall n>1}{} \Rightarrow \sum_{k=1}^{n}\frac{1}{k} = \Omega(\lg n)$

②: $\sum_{k=1}^{n}\frac{1}{k} \le 2\cdot\ln 3\cdot\lg n \underset{\forall n\ge 2}{} \Rightarrow \sum_{k=1}^{n}\frac{1}{k} = O(\lg n)$

$\Rightarrow \sum_{k=1}^{n}\frac{1}{k} = \Theta(\lg n)$

8.

$$\lg(n!) = \lg(n\times(n-1)\times\cdots\times 1) = \sum_{k=1}^{n}\lg k \le \sum_{k=1}^{n}\lg n = n\cdot\lg n \quad \text{— ①}$$

$$\lg(n!) = \sum_{k=1}^{n}\lg k = \left(\sum_{k=2}^{n}\lg k\right) + \lg 1 = \sum_{k=2}^{n}\lg k > \int_{1}^{n}\lg x\,dx \quad \text{— ②}$$

$$\int_{1}^{n}\lg x\,dx = \ln 2\cdot\int_{1}^{n}\ln x\,dx = \ln 2\cdot(n\ln n - n + 1)$$

$$\forall n\ge e^2,\ 2\cdot(n\ln n - n) = n\ln n + n(\ln n - 2) \ge n\ln n > n\lg n$$

$$\Rightarrow \int_{1}^{n}\lg x\,dx = \Omega(n\lg n) \quad \text{— ③}$$

① $\Rightarrow \lg(n!) = O(n\lg n)$

②+③ $\Rightarrow \lg(n!) = \Omega(n\lg n)$

$\Rightarrow \lg(n!) = \Theta(n\lg n)$

9.

let $m = \lfloor \lg n \rfloor$

let $a_1 = n$, $a_{k+1} = \lfloor \frac{a_k}{2} \rfloor$, then $a_{m+1} = 1$

let $b_k = a_{m+2-k}$, then $b_1 = 1$, $b_{m+1} = n$, $b_k = \lfloor \frac{b_{k+1}}{2} \rfloor$

and also $2^{k-1} \le b_k \le 2^k$

let $f_k = f(b_k)$, then $f_1 = f(1) = 1$

$$f_{m+1} = 2f_m + b_{m+1} \lg(b_{m+1})$$

$$2f_m = 4f_{m-1} + b_m \lg(b_m) \cdot 2$$

$$\vdots$$

$$+)\ 2^{m-1} f_2 = 2^m f_1 + b_2 \lg(b_2) \cdot 2^{m-1}$$

$$f_{m+1} = 2^m f_1 + \sum_{k=2}^{m+1} b_k \cdot \lg(b_k) \cdot 2^{m+1-k}$$

$$= 2^m + 2^{m+1} \cdot \sum_{k=2}^{m+1} b_k \lg(b_k) \cdot 2^{-k}$$

$$\sum_{k=2}^{m+1} 2^{k-1} \cdot \lg(2^{k-1}) \cdot 2^{-k} \le \sum_{k=2}^{m+1} b_k \lg(b_k) \cdot 2^{-k} \le \sum_{k=2}^{m+1} 2^k \cdot \lg(2^k) \cdot 2^{-k}$$

$$\sum_{k=2}^{m+1} \frac{1}{2}(k-1) = \frac{1}{2} \frac{m(m+1)}{2} = \Omega(m^2) \qquad \sum_{k=2}^{m+1} k = \frac{(m+1)(m+2)}{2} - 1 = O(m^2)$$

$$\Rightarrow 2^m(1 + 2 \cdot \Omega(m^2)) \le f_{m+1} = f(n) \le 2^m(1 + 2 \cdot O(m^2))$$

$$f(n) = 2^m(1 + 2 \cdot \Theta(m^2)) = \Theta(2^m \cdot m^2) = \Theta(n(\lg n)^2)$$

Problem 2

References:

B09902100 林弘毅

1.

```
ReverseQueue(source, helper)
    n = source.size()
    for(i=0; i<n; i++)
        for(j=0; j<i-1; j++)
            tmp = source.dequeue()
            source.enqueue(tmp)
        tail = source.dequeue()
        helper.enqueue(tail)
    for(i=0; i<n; i++)
        tmp = helper.dequeue()
        helper.enqueue(tmp)
```

2.

Because `enqueue`, `dequeue` and `size` all take $O(1)$ time, the time complexity is

$$O(1) + O(n \cdot (n+1)) + O(n) = O(1) + O(n^2) + O(n) = O(n^2)$$

3.

Use one stack (`front`) to simulate the front of the deque, and the other stack (`back`) to simulate the back.

For `push_front` and `push_back` we simply push items to the corresponding stack.

`pop_front` and `pop_back` are a bit trickier. When the corresponding stack isn't empty, we can simply pop from it. However when it's empty, we dump all items from the other stack to it, pop from it, and dump all items back.

```
push_front(deque, x)
    deque.front.push(x)

push_back(deque, x)
    deque.back.push(x)

pop_front(deque)
    if deque.front is not empty
        return deque.front.pop()
    else
        while deque.back is not empty
            deque.front.push(deque.back.pop())
        frt = deque.front.pop()
        while deque.front is not empty
            deque.back.push(deque.front.pop())
```

```
        return frt

pop_back(deque)
    if deque.back is not empty
        return deque.back.pop()
    else
        while deque.front is not empty
            deque.back.push(deque.front.pop())
        bck = deque.back.pop()
        while deque.back is not empty
            deque.front.push(deque.back.pop())
        return bck
```

4.

Because `stack.push()` takes $O(1)$ time, the time complexity of `push_front()` is $O(1)$.

5.

Because `stack.push()` takes $O(1)$ time, the time complexity of `push_back()` is $O(1)$.

6.

Let n be the length of the deque. When `deque.front` is not empty, time complexity of `pop_front()` $=$ time complexity of `stack.pop()` $= O(1)$.

When `deque.front` it empty, dumping items from `deque.back` to `deque.front` takes $O(n)$ time, `stack.pop()` takes $O(1)$ time, and dumping items from `deque.front` back to `deque.back` takes another $O(n)$ time. Therefore the total time complexity of `pop_front()` is $O(n) + O(1) + O(n) = O(n)$. The performance of my implementation tends to be better if `push_front` and `pop_front` are more balanced.

7.

Since the algorithm I have for `pop_back` is basically the same as `pop_front`, they share the same time complexity, that is $O(1)$ for best and $O(n)$ for worst.

8.

The best case happens when the stack was never full during n pushes, the time complexity in this case is $n \cdot O(1) = O(n)$.

For the worst case, it happens when we start pushing from 0 element to $3^k = n$ elements, because it will trigger `enlarge()` most times. In this case, the `S->arr[++S->top] = data;` part has the same time complexity, which is $O(n)$, therefore we only needs to look at how much time complexity do all `enlarge()` add.

`enlarge()` will happen when there is $1,\ 3,\ 3^2, \cdots, 3^k$. And the enlarged size would be $3^1,\ 3^2,\ 3^3, \cdots, 3^{k+1}$.

The time complexity for that would be $O(3^1 + 3^2 + \cdots + 3^{k+1}) = O(\frac{3(3^{k+1}-1)}{3-1}) = O(3^k) = O(n)$

Therefore, the total time complexity for worst case would also be $O(n) + O(n) = O(n)$.

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 $O(1)$.
