Deadline: December 5th (Monday)
December 6th / December 8th ---> presentation
Exam II: Up to and including stacks and queues (No inheritance, no polymorphism, no exceptions)

- Stack : LIFO (Last In First Out) ---> scheduling discipline
Stack ADT: size(), isEmpty(), top, pop, and push
ArrayBasedStack implements Stack
Grouping symbol matching problem: validating an expression that contains n tokens
Brute force solution: Time complexity O(n^2); Space complexity: O(1)
Stack: Time complexity O(n) and Space complexity O(n)

- Queue: FIFO (First In First Out)
Elements can be inserted at any time. However, only the one that spent the longest time
in the queue can be removed at any given instant.

1) Rear end of the queue; 2) Front element
Queue ADT:
a. int size()
b. boolean isEmpty()
c. Object front() throws QueueException
d. Object dequeue() throws QueueException
e. void enqueue(Object obj) throws QueueException

Approach#1: Use a regular array to realize the queue data structures. If we have n elements
in the queue, they have to occupy the first n positions of the queue.

Where should I store the front element?
Option#1: Place the front element at index 0

P1 P2 P3 P4

Inserting an element can be done in O(1) time
Removing an element would require O(n) time (shifting)

P2: 0 P3: 1 P4: 2

Option#2: Place the front element at index n - 1

Dequeue operation is O(1)
Enqueue is O(n) operation

P4: 0 P3: 1 P2: 2 P1: 3

Conclusion: We should not use a regular array to implement the queue

Approach#2:

Let the content of the queue drift within the array ===> Use a drifting array to create the queue

Introduce two additional variables:
1. front: index of the front element if the queue is not empty
2. rear: index of the next available cell in the array

Queue is empty ===> front = rear = 0 (Initial condition)

P1 P2 P3
N = 4 = capacity of the queue

array[rear] = P2
rear = rear + 1
array[front] = null
front = front + 1

null: 0 null: 1 null: 2 null: 3 front = rear = 4

Problem: We ran out of space although the queue is not full
Conclusion: By letting the queue be a drifting one, we were able to solve one problem. However, we encountered the one highlighted above.

front = 3 ===> front = front + 1 % array.length

Approach#3: Use a circular drifting array

P1: front=0=rear P2 P3 P4

It will be hard to differentiate an empty queue from a full queue
size = track the size of the queue

Prime numbers ---> distribution
[a; b]

41 soldiers ---> Josephus
41 = 32 + 9 ===> Winning position: 2*9 + 1 = 19

k = 1

N = 2^a + b ---> Position = 2*b + 1

X ---> Largest power of 2 that divides X ---> O(1)

k = 1

C T R

Time complexity: O(Nk)
Space complexity: O(N)