Assignment 1 -Data Structure - 25 Spring

Exercise 1 : Correctness

1. Specification for REVERSE_SEQUENCE

```
REV(A, B):
    # asserts that B is the reverse of A
    if length(A) == 0 or length(B) == 0:
        return 1
    if length(A) != length(B):
        return 0
    else:
        n = length(B)
        i = 0
        while(i < n):
            if A[i] != B[n-i-1]:
                return 0
            i = i + 1
        return 1
```

Contract:

Pc : |A| = n && n % 2 = 0

Pc' : |A| = n && n % 2 = 0 && i = 0

Inv : $0 \leq i \leq n$ && REV(A_0[: i-1], A[n-i :]) && REV(A_0[n-i :], A[: i-1]) && A[i : n-1-i] = A_0[i : n-1-i]

Qc' : i = $\frac{n}{2}$ && REV(A_0, A)

Qc : REV(A_0, A)

2. Correctness Proof

Checking correctness is to certify the following conditions hold :

1. **Initialization :** Assume `Pc'` holds, we want to show it implies `Inv` hold:

   Here, we have no swapping yet, so A = A_0

   - From `Pc'`, i = 0, then, $0 \leq i \leq n$
   - From `Pc'`, i = 0, then, REV(A_0[: i-1], A[n-i :]) = REV(A_0[: -1], A[n :]),

     by our definition, since A[n :] is an empty sequence, REV(A_0[-1], A[n]) = 1
   - From `Pc'`, i = 0, then, REV(A_0[n-i :], A[: i-1]) = REV(A_0[n :], A[: -1]),

     similarly, since A_0[n :] is empty, REV(A_0[n :], A[: -1]) = 1
   - Since we have no swapping yet, A[0 : n-1] = A = A_0 = A_0[0 : n-1]

2. **Maintenance :** Assume `Inv&&B` hold, we want to prove the loop make `Inv` true again

   `Inv&&B` = { $0 \leq i \leq \frac{n}{2}$ && REV(A_0[: i-1], A[n-i :]) && REV(A_0[n-i :], A[: i-1])

   && A[i : n-1-i] = A_0[i : n-1-i]}

   Name i_0 the old value of i before a new loop start.

   After one loop, $i = i_0 + 1$ and swap(A[i_0], A[n-1-i_0]).

   - Since $0 \leq i_0 \leq \frac{n}{2}$, then $0 \leq i_0 + 1 \leq n$, equivalent to $0 \leq i \leq n$.

- REV(A_0[: i_0-1], A[n-i_0 :]) && A_0[i_0] == A[n-1-i_0] imply REV(A_0[: i_0], A[n-1-i_0 :]), equivalent to REV(A_0[: i -1], A[n-i:])
- REV(A_0[n-i_0 :], A[: i_0-1]) && A[i_0] == A_0[n-1-i_0] imply REV(A_0[n-1-i_0 :], A[: i_0]), equivalent to REV(A_0[n-i :], A[: i-1])
- A[i_0 : n-1-i_0] = A_0[i_0 : n-1-i_0] before the loop, but A[i_0] $\neq$ A_0[i_0] and A[n-1-i_0] $\neq$ A_0[n-1-i_0] after the loop, so that only A[i : n-1-i] = A_0[i : n-1-i] remains

Thus, `Inv` will be true after any loop in the algorithm.

3. **Termination :** Assume `Inv&&(not B)` hold, we want to show `Qc` hold:

`Inv&&(not B)` = {n/2 <= i <= n && REV(A_0[: i-1], A[n-i :]) && REV(A_0[n-i :], A[: i-1])

&& A[i : n-1-i] = A_0[i : n-1-i]}

When it jumps out from the loops, i occurs to be n/2,

and REV(A_0[: n/2 -1], A[n/2 :]) && REV(A_0[n/2 :], A[: n/2 -1]).

By the property of REV(A, B), we can obtain REV(A_0, A).

3. For sequence of odd length:

Yes. The algorithm remains correct when the length of A is odd.

Says n is odd, then n/2 -1 is the last i that will be considered in the loop. m = n/2 will jump out the loop, but notice that: 2*m +1 = n, m = n-m-1, so A[m] = A[n-m-1] trivially. There is no need to swap the same element in a sequence. And the element A[m] will preserve its position, independent with the reverse operation on the whole sequence.

Exercise 2

Exercise 3

1. Define the ADT `DoubleStackOfInt`

```
public interface DoubleStackOfInt {
    void push(int elem) // an integer onto the stack.
    void pop()          // an integer from the stack.
    boolean empty()     // checks the stack is empty.
    int top()           // consult the top element without popping it.
}
```

2.

6.

Exercise 4

1. Complexity Analyses

(a). add(int) ⇒ O(1)

```java
public void add(int newAttack) {
    if ( newAttack <= 0 ) {
        throw new RuntimeException("New Attack is not positive.");
    }
    attacksQueue.enqueue(newAttack);
}
```

The function `add(int)` performs the following operations:

- A conditional check `if(newAttack <= 0)`, runs in O(1);
- An exception throw if the given power is not positive, runs in O(1);
- An operation `enqueue` defined under Queue, in which:

  ```java
  public void enqueue(int elem) {           // passing parameter --> O(1)
      Node newnode = new Node(elem);        // the constructor, rus in O(1)
      if ( head == null && tail == null ) { // simple conditional check --> O(1)
          head = newnode;                   // if the queue is empty, then
          tail = newnode;                   // set head and tail to newnode -->O(1)
      } else {
          tail.next = newnode;              // otherwise,
          tail = newnode;                   // update tail and tail.next --> O(1)
      }
      count ++;
      sum += elem;                          // basic arithmetic --> O(1)
  }                                         // IN TOTAL, `enqueue` RUNS IN O(1)
  ```

  so `enqueue` also runs in O(1).

In total, the time complexity of `add(int newAttack)` is **O(1)**.

(b). boolean alive( ) ⇒ O(1)

```java
public boolean alive() {
    return !shieldsQueue.isEmpty();
}
```

The `boolean alive()` function performs the following operations:

- The operation `isEmpty` defined under Queue, in which:

  ```java
  public boolean isEmpty() {
      return head == null && tail == null;//check if both head, tail are null--> O(1)
  }
  ```

  so `isEmpty` runs in O(1);

- `!` inverts the boolean result from `isEmpty` ⇒ O(1);

In total, the time complexity of `boolean alive ( )` is **O(1)**.

(c). void addShield(int) ⇒ O(1)

```
public void addShield(int newShield) {
    if ( newShield <= 0 ) {
        throw new RuntimeException("addShield(): New shield is not positive.");
    }
    shieldsQueue.enqueue(newShield);
}
```

The `void addShield(int)` function performs the following operations:

- A conditional check `if(newShield <= 0)` , runs in O(1);
- An exception throw if the given shield is not positive, runs in O(1);
- The operation `enqueue` defined under Queue, we already proved its time complexity is O(1);

so in total, the time complexity of `void addShield(int)` is **O(1)**.

(d). boolean repel(int) ⇒ O(S)

```
private boolean repel(int anAttack) {
    if ( anAttack <= 0 ) {
        throw new RuntimeException("repel(): Attack is not positive");
    }

    while ( !shieldsQueue.isEmpty() && anAttack > 0 ) {

        int topShield = shieldsQueue.front();

        if ( anAttack > topShield ) {
            anAttack -= topShield;
            shieldsQueue.dequeue();
        } else if ( topShield > anAttack ) {
            shieldsQueue.setFront(topShield - anAttack);
            anAttack = 0;
        } else {
            shieldsQueue.dequeue();
            anAttack = 0;
        }
    }

    return alive();
}
```

The `boolean repel(int)` function performs the following operations:

- A conditional check `if(anAttack <= 0)` , runs in O(1);
- An exception throw if the given power is not positive, runs in O(1);
- A while loop, continues until the shield queue comes to empty or the attack drops to 0. So the worst case will be no shields left, which means the loop circulating S times, that is :
    - **S+1** times conditional check, containing logic negation, `isEmpty` and basic comparison, all in O(1);

    - S times update topShield, calling the operation `front` defined under Queue, runs in O(1):

```
public int front() {
    if ( isEmpty() ) {   // `isEmpty` runs in O(1), already proevd
        throw new RuntimeException("Queue.front(): The queue is empty.");
    }                    // exception throw --> O(1)
    return head.data;    // return int --> O(1)
}                        // IN TOTAL, `front` RUNS IN O(1)
```

    - S times conditional check, contains basic comparison between int, runs in O(1);
    - S times call operation `dequeue` or `setFront`, both run in O(1):

```
public int dequeue() {
    if ( isEmpty() ) {           // `isEmpty` runs in O(1), already proevd
        throw new RuntimeException("Queue.dequeue(): The queue is empty.");
    }                            // exception throw --> O(1)
    int popedElem = head.data;   // initialize int --> O(1)
    if ( head == tail ) {        // conditional check with boolean --> O(1)
        head = null;
        tail = null;             // update head and tail --> O(1)
    } else {
        head = head.next;        // move head --> O(1)
    }
    count --;
    sum -= popedElem;            // basic arithmetic --> O(1)
    return popedElem;            // return int --> O(1)
}                                // IN TOTAL, `dequeue` RUNS IN O(1)
```

```
public void setFront(int newData) { // passing parameter --> O(1)
    if ( isEmpty() ) {              // `isEmpty` runs in O(1), already proevd
        throw new RuntimeException("Queue.setFront(): The queue is empty.");
    }                               // exception throw --> O(1)
    sum += newData - head.data;
    head.data = newData;            // basic arithmetic --> O(1)
}                                   // IN TOTAL, `setFront` RUNS IN O(1)
```

- Return the boolean value get from `boolean alive()`, which runs in O(1) as we already know;

so in total, the time complexity of `boolean repel(int)` is

$$O(1) + O(1) + (S+1) \times O(1) + S \times (O(1) + O(1) + O(1)) + O(1) = (2S+4) \times O(1) = O(2S+1) = O(S)$$

(e). boolean repel(AttackStrategy) ⇒ O(S+A)

```
public boolean repel(AttackStrategy anAttackStrategy) {
    AttackStrategyImplementation strategy = (AttackStrategyImplementation) anAttackStrategy;
    if ( strategy.isEmpty() ) {
        throw new RuntimeException("repel(): Attack Strategy is empty.");
    }
    while ( alive() && !strategy.isEmpty() ) {
        repel( strategy.popAttack() );
    }
    return alive();
}
```

The `boolean repel(AttackStrategy)` function performs the following operations:

- Reset the input anAttcakStrategy as strategy, to better access the mothods, which runs in O(1);
- A conditional check with `isEmpty`, which runs in O(1);
- An exception throw if the given attack strategy is empty, runs in O(1);
- A while loop, continues until the warrior has no more shields or the enemy has no more attacks. So the worst case will be: when the enemy applies the last attack, the innermost shield of the warrior breaks. In a more specific situation under this case that attack and shield update at different time, the loop circulates **S+A** times, that is:
  - **S+A+1** times conditional check, containing `!`, `boolean alive()` and `isEmpty`, all run in O(1);
  - S+A times call function `popAttack()` defined in AttackStrategy, in which:

    ```
    public int popAttack() {
        if ( isEmpty() ) {             // conditional check with `isEmpty` --> O(1)
            throw new RuntimeException("Attack strategy is empty.");
        }                              // exception throw --> O(1)
        return attacksQueue.dequeue(); // return int given by `dequeue` --> O(1)
    }                                  // IN TOTAL, `popAttack` RUNS IN O(1)
    ```

    so `int popAttack()` runs in O(1);
- Return the boolean value get from `boolean alive()`, which runs in O(1) as we already know;

so in total, the time complexity of `boolean repel(AttackStrategy)` is

$$O(1) + O(1) + (S + A + 1) \times O(1) + (S + A) \times O(1) + O(1) = (2S + 2A + 4) \times O(1) = O(2S + 2A + 4) = O(S + A)$$

(f). int shields( ) $\Rightarrow$ O(1)

```
public int shields() {
    return shieldsQueue.elemNum();
}
```

The `int shields( )` function performs the following operations:

- Return the int value given by `elemNum`, which is defined under Queue,

  ```
  public int elemNum() {
      return count;    // return the value of an attribute --> O(1)
  }
  ```

  so `elemNum` runs in O(1);

Thus, the time complexity of `int shields( )` is **O(1)**.

(g). int remainingPower( ) $\Rightarrow$ O(1)

```
public int remainingPower() {
    return shieldsQueue.elemSum();
}
```

The `int remainingPower( )` function performs the following operations:

- Return the int value given by `elemSum`, which is defined under Queue,

```
public int elemSum() {
    return sum;      // // return the value of an attribute --> O(1)
}
```

so `elemSum` runs in O(1);

Thus, the time complexity of `int remainingPower( )` is **O(1)**.

2. Representation Invariant for `Warrior` ADT

$R: \langle \text{head: } Node,\ \text{tail: } Node,\ \text{count: } int,\ \text{sum: } int \rangle$

$Node: \langle \text{data: } int, \text{next: } Node \rangle$

$RI(R) = R$ is acyclic && tail is reachable from head && (tail == null || tail.next == null)

&& count = the number of shields of the warrior $\geq$ 0

&& sum = the sum of all values of the shields $\geq$ 0