Data Structures 1 – Spring 2025 - Assignment A1

version 1.1

Rules:

- **Submission Deadline: April 11 at 13:00**. Late submissions will incur the following penalties:
  - **April 11 (after 13:00) – April 12:** 10-point deduction
  - **April 13:** 20-point deduction
  - **April 14:** 30-point deduction

  Submissions will not be accepted after April 14.
- The assignment should be completed in groups of two students (already assigned in Moodle).
- Students will be randomly selected to provide an oral explanation of their solution to the exercises.
- Questions regarding the exercise statements are welcome during office hours. However, teachers will not assess your solution.
- There is no guarantee that questions submitted via email will receive a response. Should you decide to submit questions by email anyway, you must address both professors: `herman.schinca@gtiit.edu.cn` and `hernan.melgratti@gtiit.edu.cn` . **Emails sent to only one professor will not be answered and no notification will be provided**.
- The code must be implemented in Java, compiled in VSCode, and pass the developed tests. This is the minimum requirement. Note that graders and professors may run additional tests. Furthermore, passing tests is not a sufficient condition for correctness. The quality of the submitted code will also be graded.
- Submit **one zip file containing the code for each exercise** (wet part) and **one PDF for all other exercises** (dry part).

Plagiarism and Academic Misconduct:

In cases of copying or plagiarism, exams will be placed under review. If plagiarism is confirmed, the case will be reported to the GTIIT Discipline Committee. **This includes submitting solutions found online or generated by AI tools.** All group members will be held accountable for any submission that violates this policy.

Exercise 1: Correctness [20 points]

A **reverse of a given sequence** `A` is a sequence in which the elements appear in the opposite order compared to their original arrangement in `A` . If a sequence is given as:

$A = [a_0, a_1, a_2, ..., a_n]$

Then, the reverse sequence is:

$REVERSE(A) = [a_n, a_{n-1}, ..., a_1, a_0]$

Algorithm

Consider the following algorithm for reversing a given sequence `A`, assuming that `n` is the size of the sequence (the sequence is indexed from `0` to `n - 1`):

```
REVERSE_SEQUENCE(A, n)
    i = 0;
    while (i < n div 2):
        if (A[i] != A[n - 1 - i]):
            temp = A[i];
            A[i] = A[n - 1 - i];
            A[n - 1 - i] = temp;
        i = i + 1;
```

Tasks

1. **Specification for** `REVERSE_SEQUENCE`: Formally define the contract for `REVERSE_SEQUENCE` by requiring the length `n` to be even.

   Hint: Define a predicate `REV(A, B)` that asserts that `B` is the reverse of `A`.

2. **Correctness Proof**: Show that the algorithm correctly computes the reverse of a sequence of even length.

   You can assume the following properties on `REV(A, B)`:

   - REV(A,B) → REV(B,A)
   - REV(A[a_1 .. a_2], B[b_2 + 1 .. b_3]) && REV(A[a_2 + 1 .. a_3], B[b_1 .. b_2]) → REV(A[a_1 .. a_3], B[b_1 .. b_3])
   - REV(A[a_1 .. a_2-1], B[b_1 + 1 .. b_2]) && A[a_2] == B[b_1] → REV(A[a_1 .. a_2], B[b_1 .. b_2])

3. **Handling Sequences of odd length**: Determine whether the algorithm remains correct when the length of `A` is odd.

   - If **yes**, update the correctness proof accordingly.
   - If **no**, identify where the algorithm fails and explain why.

Exercise 2: Time Complexity [15 points]

Consider the following algorithm, which takes a sequence `A`, the size `n` of `A`, and an integer `h` such that `h <= n`. The algorithm also uses another function, `AUXILIARY`, which takes a sequence and an integer `k`, where `k` is less than or equal to the sequence's size.

Version 1.1: You can assume `h > 0` and `k >= 0`.

The algorithm assumes that sequences of length `n` are indexed from `0` to `n-1`.

```
ALGORITHM(A, n, h)
    for (k = 0; k < n / h; k++)
        AUXILIARY(A[k*h .. n-1], h);    //version 1.1: note n-1
    AUXILIARY(A[k*h .. n-1], n - k*h); //version 1.1: note n-1

    for (k = h; k < n; k *= 2)
        for (j = 0; j < n - k; j += 2*k)
            MERGE(A[j .. j+k-1], A[j+k .. min(j+2*k-1, n-1)]);
```

Tasks

1. **Determine the time complexity in the worst case** of the algorithm, expressed in **Big-O notation**, assuming that the complexity of `AUXILIARY(A, h)` is **$\Theta(h^2)$**. The complexity should be expressed as a function of `n` and `h`.

   Note: Assume that parameters in the call to `AUXILIARY` are passed by reference. Therefore, in `AUXILIARY(A[k*h .. n], h)` the sequence `[k*h .. n]` is not copied, and parameters passing is `O(1)`.

2. **Analyze the complexity** for the case where `h = 1`.

3. **Analyze the complexity** for the case where `h = n`.

4. **Implement the algorithm in Java.** In your implementation, `AUXILIARY` should be implemented using **Insertion Sort**. To avoid copying the sequence, you can define the function:

   ```
   void insertionSort(int[] a, int low, int size)
   ```

   where `low` is the starting index, and `size` is the number of elements to be processed. Then, calls like `AUXILIARY(A[k*h .. n], h)` can be replaced with `insertionSort(A, k*h, h)`.

   Hint: The resulting algorithm is a **sorting algorithm** that optimizes merge sort by taking advantage of the lower constant factors of insertion sort when sorting small sequences. Use this information to verify the correctness of your answers in **items 1, 2, and 3**.

Exercise 3: Abstract Data Type [25 points]

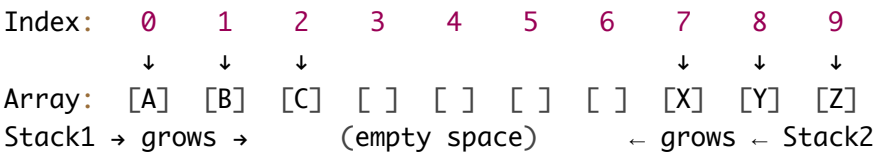
A `DoubleStackOfInt` is a data structure that consists of **two stacks**, which allows independent stack operations.

Tasks

1. **Define the ADT** `DoubleStackOfInt` Provide a Java interface for `DoubleStackOfInt`. The interface should support the following operations **for each stack**:

   - **push** an integer onto the stack.
   - **pop** an integer from the stack.
   - **empty** checks the stack is empty.
   - **top** consult the top element without popping it.

2. **Implementation** A `DoubleStackOfInt` can be implemented using an array, where the two stacks are positioned at **opposite ends** of the array and grow **toward each other**.



Define the structure used for representing a `DoubleStackOfInt` **of integer values** with the restriction that elements should be stored in **one** array and ensuring that all operations run in **O(1)** worst-case time complexity. Clearly define the **representation invariant** and **abstraction function** for your implementation. Show that the provided structure satisfies the time complexity requirements.

3. **Define** `DoubleStackOfIntOnArray` in **Java**, which implements the interface `DoubleStackOfInt` by using the representation defined in the previous item.

4. **Provide meaningful test cases**. Write test cases covering different scenarios, such as:

   - Pushing and popping elements from both stacks.
   - Handling an **empty stack**.
   - Handling a **full stack** (when the structure reaches its maximum capacity).
   - Popping from a **non-empty but non-full stack**, etc.

   The quality of the provided tests will be evaluated.

5. **Implement a sorting algorithm using** `DoubleStackOfIntOnArray` . Use a `DoubleStackOfIntOnArray` to **sort elements** placed in one of the stacks. The second stack is used temporarily to store elements. The sorting algorithm follows these steps:

   - If the stack containing the original elements is **empty**, move all elements from the second stack back to the original stack (since they are already sorted in descending order), and terminate.
   - Otherwise, compare the **top element** of the original stack with the **top element** of the secondary stack:
     - If the top element of the original stack is **greater**, move it to the secondary stack.
     - If it is **smaller**, remove it, move all elements from the secondary stack that are **greater** back to the original stack, and then push the removed element onto the secondary stack.
   - Repeat the process until all elements are sorted in the original stack in **ascending order**.

   Note that while the algorithm is running, the second stack maintains elements in **descending order**.

IMPORTANT: Although you are expected to submit a single Java project containing the solution to all programming tasks in this exercise, the source code in your solution should be split into at least five files:

- One defining the `DoubleStackOfInt` interface,
- Another for `DoubleStackOfIntOnArray` ,
- One containing tests for `DoubleStackOfIntOnArray` ,
- Another for the sorting algorithm,
- And one for the tests of the sorting algorithm.

6. **Determine the time complexity** of the sorting algorithm and justify your answer.

Exercise 4: Galactic Siege: The Last Line of

Defense [40 points]

Problem Description

You are required to implement the data structures and functionalities of a game featuring a cybernetic warrior defending against enemy assaults in space. The survival of a warrior depends on **energy shields**, each with a specific **defense power**. The enemy devises **attack strategies**, which consist of sequences of attacks designed to break through the defenses.

Game Mechanics

- A warrior has a **layered structure of shields**, similar to onion peels, each with a defense power. For example, `[100, 75, 50]`, where the **outermost shield** has `100` defense power and the **innermost shield** has `50`.
- An **attack strategy** is a sequence of attack values (e.g., `[30, 40, 80, 60]`), where each value represents the damage of an attack.
- The **attack sequence** is applied **one attack at a time** to the outermost shield:
  i. If the shield's power is **greater than** the attack damage, the shield **absorbs the hit**, but its power is reduced.
  ii. If the shield's power is **less than or equal to** the attack damage, the shield is **destroyed**, and the attack moves to the next shield.
  iii. The process continues until either **all shields are destroyed** (**successful attack**) or **all attacks are used** but the warrior still has shields (**unsuccessful attack**).

Example Scenarios

Example 1: Unsuccessful Attack

Shields: `[100, 75, 50]` **Attacks:** `[30, 40, 30, 60]` -- Version 1.1: change power of 3rd attact from 80 to 30

Process:

- Attack `30` → Top shield (`100 → 70` left)
- Attack `40` → Top shield (`70 → 30` left)
- Attack `30` → Top shield destroyed (`30 <= 30`, remove it)
- Attack `60` → Next shield (`75 → 15` left)

Output: `"Attack Unsuccessful"` (warrior still has shields left)

Example 2: Successful Attack

Shields: `[10, 50]` **Attacks:** `[40, 25]` -- Version 1.1: removed second attack of power 30 .

Process:

- Attack `40` → Top shield destroyed (`10 < 40`, remove it and remaining power for attacking next shield)
  Version 1.1: see remaining power is used for next shield
- Attack `30` → Next shield (`50 → 20` left)

- Attack `25` → Next shield destroyed (`20 < 25` , remove it)

Output: `"Attack Successful"` (all shields are destroyed)

Objective

Define and implement required data types that allow a warrior to equip shields, build attack strategies, and simulate the defense mechanism to determine whether an attack strategy successfully breaches the warrior's defenses.

Your solution should be **modular**. Specifically, you must identify appropriate **data structures** (such as lists, queues, or stacks) and implement the necessary abstractions. For example, if the warrior's shields are stored in a list, then **list operations should not be part of the warrior's logic—**instead, the warrior should interact with the list through the corresponding interface.

The following is the **minimal set of required operations** for each abstraction. However, you may need to implement **additional operations with appropriate complexity** to support these functionalities. In all cases, complexity refers to worst-time runtime complexity.

Attack Strategy (`AttackStrategy`)

- `add(int)` (`O(1)`) Adds a new attack with the given power to the strategy.
  - The new attack is applied **after** all existing attacks in the strategy.
  - The operation fails by throwing a suitable exception if the given power is **not positive**.

Warrior (`Warrior`)

In the following, `S` represents the number of shields a warrior possesses, and `A` represents the number of attacks in the applied attack strategy.

- `boolean alive()` (`O(1)`) Returns whether the warrior is alive, i.e., whether they still have shields.
- `void addShield(int)` (`O(1)`) Adds an **outer shield** with the given power to the warrior.
  - The implementation should ensure that power is **positive**. If it is not, the operation should fail by throwing a suitable exception.
- `boolean repel(int)` (`O(S)`) Applies an attack of the given power to the warrior.
  - Shields are **removed** until the attack is repelled.
  - Returns `true` if the attack is repelled, `false` if the warrior is **defeated**.
  - This method **modifies the warrior**, meaning that after the attack, only **remaining shields** are kept.
- `boolean repel(AttackStrategy)` (`O(S + A)`) Applies all attacks in the given strategy, one by one.
  - The process stops if an attack **cannot be repelled**.
  - Returns `true` if the **entire** strategy is repelled, `false` otherwise.
  - This method **modifies the warrior**, removing destroyed shields.
- `int shields()` (`O(1)`) Returns the **number of shields** the warrior has left.
- `int remainingPower()` (`O(1)`) Returns the **sum of the power of all the shields** the warrior has.

Tasks

1. Design the data structures and algorithms to support the defined functionality while meeting the required runtime complexity. Provide a complexity analysis of the implemented methods to demonstrate that they satisfy the required complexity. When choosing the appropriate data structures note that the number of shields a warrior can have and the number of attacks in a strategy are unbounded.
2. Give the representation invariant for the `Warrior` ADT.
3. Implement your solution by completing the provided Java project template. You should implement `AttackStrategy` and `Warrior`, along with any supporting data structures, observing the following additional requirements:
    - If your implementation requires collections such as lists, stacks, or queues, you must define and implement them yourself. Your solution cannot use any collection provided by Java (e.g., the Collections library) or any other external implementation. The use of basic arrays (e.g., `int[]`, `double[]`, etc.) is allowed.
    - Your source code should be modular, meaning you should provide separate source files for `AttackStrategy`, `Warrior`, and any collections you define. Additionally, the interface for each auxiliary data structure should be provided in a separate file.
4. Provide tests for both the collections you have implemented and the ADTs `AttackStrategy` and `Warrior`. The quality of the tests you provide will be evaluated.