# Components of the L1 assessment

L1 intro - 2 points

L1 mandatory part - 6 points

L1 optional part - 2 points

# L1 Task

## L1 Intro - an Introductory Exercise (2 points)

1. Familiarize yourself with the programming code in the `l1-intro` project.
2. Implement and test the unimplemented methods of the `l1-intro` project.

## L1 Mandatory Part (6 points)

1. Familiarize yourself with the programming code in the `l1-assignment-linear-data-strucutres` project.
2. Implement and test the unimplemented methods of the `l1-assignment-linear-data-strucutres` project. **(2 points)**
3. Create a `Stack<E>` interface that has the following abstract methods:
   a. `E pop()` - for deleting and returning the top element in the stack (computational complexity O(1)).
   b. `void push(E item)` - for inserting a new element at the top of the stack (computational complexity O(1) amortized).
   c. `E peak()` - for returning the first element in the stack (computational complexity O(1)).
   d. `boolean isEmpty()` - to check if the stack is empty (computational complexity O(1)).
4. Create two classes, `ArrayStack<E>` and `LinkedListStack<E>`. Both of these classes must implement the `Stack<E>` interface. `ArrayStack<E>` implements the stack data structure on the basis of an array (the Java collection class `ArrayList<E>` can be used to implement an array), `LinkedListStack<E>` implements the stack data structure on the basis of a linked list (the generic `LinkedList` class contained in `Generic_Lists` can be used to implement a stack or the Java collection class `LinkedList<E>` can be used). Test the operations of these data structures. **(2 points)**
5. Create an interface `Queue<E>` that has these abstract methods:
   a. `void enqueue(E item)` - for adding a new element to the end of the queue (computational complexity O(1) amortised).
   b. `E dequeue()` - for deleting and returning the first element in the queue (computational complexity O(1) amortized).
   c. `E peak()` - for the return of the first element in the queue (computational complexity O(1)).
   d. `boolean isEmpty()` - to check if the queue is empty (computational complexity O(1)).
6. Create two classes, `ArrayQueue<E>` and `LinkedListQueue<E>`. Both classes must implement the interface `Queue<E>`. `ArrayQueue<E>` implements the queue data structure on the basis of an array (you can use the system Java collection class `ArrayList<E>` to implement an array), `LinkedListQueue<E>` implements the queue data structure on the basis of a linked list (you can use either the `GenericList` class in `l1-assignment-linear-data-strucutres` or the Java collection class `LinkedList<E>`). Test the operations of these data structures. **(2 points)**

# L1 optional part (2 points)

Solve the following problems:

1. **Task 1 (1 point).** We have a string of length N (0 < N <= 1000). The string consists of bracket symbols "{", "(", "[", "[", "]", ")" and "}". Implement an algorithm to check if the brackets are balanced. The string of brackets is considered to be balanced if:
    a. Open brackets must be closed by the same type of brackets.
    b. Open brackets must be closed in the correct order.
    c. Every close bracket has a corresponding open bracket of the same type.

    Consider the following examples:

    "{(})" – is unbalanced since open brackets are closed in incorrect order.

    “[” – is unbalanced since open bracket is not closed.

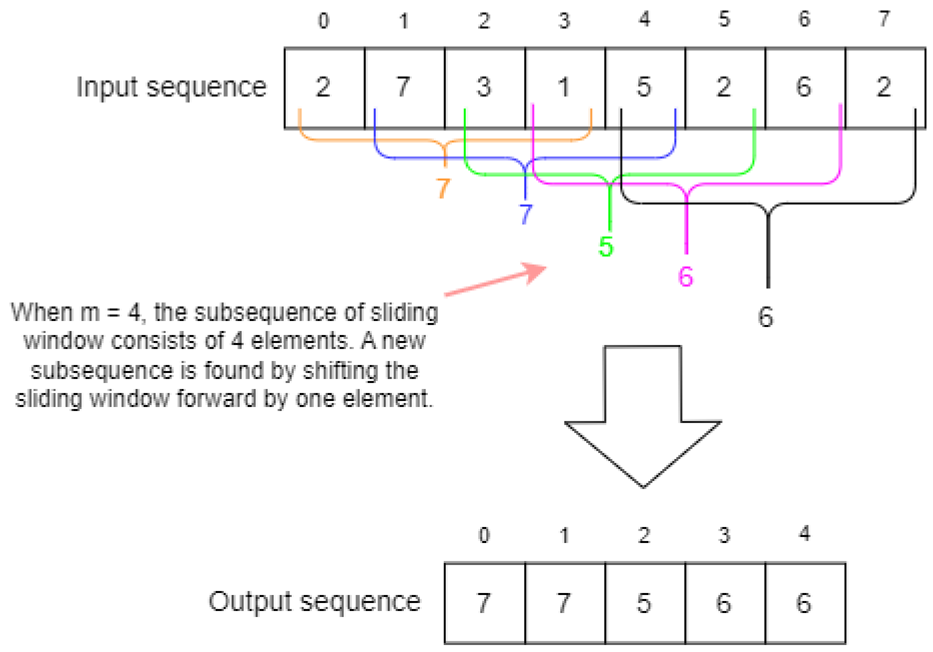
    "{()}{[]}" – is balanced.

    The solution must return a boolean denoting whether brackets are balanced or not. **The computational complexity of the algorithm must be O(n).**

    Input/Output samples:

    | Input | Output |
    |---|---|
    | "}" | False |
    | "{()}{[]}" | True |
    | "[{}}" | False |
    | "{()[{}]}{}" | True |
    | "{(})" | False |
    | "([(]{)})" | False |

2. **Task 2 (1 point).** We have a sequence of integers $a_1, a_2, \dots, a_n$. Here n (1 ≤ n ≤ $10^5$) denotes the length of the sequence. The number m (1 ≤ m ≤ n) is the size of sliding window, i.e. subsequence. At first, the sliding window occupies first m positions of the sequence (i.e. sliding window starts form 0 to m-1), then it keeps on shifting forward by one position until the end of the sliding window reaches the end of the sequence. After each shift, including the start position, we search the maximum element within the subsequence. An example is given below for m = 4:



0
1
2
3
4
5
6
7
Input sequence
2
7
3
1
5
2
6
2
7
7
5
6
6
When m = 4, the subsequence of sliding window consists of 4 elements. A new subsequence is found by shifting the sliding window forward by one element.
0
1
2
3
4
Output sequence
7
7
5
6
6

Your task is to implement an algorithm, which calculates maximum element of each subsequence bounded by sliding window. The naive solution of this problem is O(nm). <u>**You need to implement an algorithm that ensures a computational complexity of O(n)**</u>**.**

Input/Output samples:

| Input | Output |
|---|---|
| k = 4<br>[2 7 3 1 5 2 6 2] | [7 7 5 6 6] |
| k = 1<br>[2 7 3 1 5 2 6 2] | [2 7 3 1 5 2 6 2] |
| k = 8<br>[2 7 3 1 5 2 6 2] | [7] |

## L1 Assessment

During the assessment, lecturers may ask students to complete the following tasks and answer the questions below. <u>The list of questions and exercises is not exhaustive - lecturers may ask questions or practical exercises related to the topic during the defence that are not included in this list.</u>

## Possible practical exercises from the L1 mandatory part

Implement one or more of the methods from the following list:

| **Returned type** | **Description** |
|---|---|
| **void** | **add(int index, E element)** |
| | Inserts the specified element at the specified position in this list. |
| **boolean** | **addAll(LinkedList<? extends E> c)** |
| | Appends all of the elements in the specified collection to the end of this list, in the order that they are returned by the specified collection's iterator. |
| **boolean** | **addAll(int index, LinkedList<? extends E> c)** |

| | |
|---|---|
| | Inserts all of the elements in the specified collection into this list at the specified position. |
| **boolean** | **contains(Object o)** |
| | Returns true if this list contains the specified element. |
| **boolean** | **containsAll(LinkedList<?> c)** |
| | Returns true if this list contains all of the elements of the specified collection. |
| **boolean** | **equals(Object o)** |
| | Compares the specified object with this list for equality (List equivalence) |
| **int** | **indexOf(Object o)** |
| | Returns the index of the first occurrence of the specified element in this list, or -1 if this list does not contain the element. |
| **int** | **lastIndexOf(Object o)** |
| | Returns the index of the last occurrence of the specified element in this list, or -1 if this list does not contain the element. |
| **E** | **remove(int index)** |
| | Removes the element at the specified position in this list. |
| **boolean** | **remove(Object o)** |
| | Removes the first occurrence of the specified element from this list, if it is present. |
| **boolean** | **removeAll(LinkedList<?> c)** |
| | Removes from this list all of its elements that are contained in the specified collection. |
| **boolean** | **retainAll(LinkedList<?> c)** |
| | Retains only the elements in this list that are contained in the specified collection. |
| **E** | **set(int index, E element)** |
| | Replaces the element at the specified position in this list with the specified element. |
| **List<E>** | **subList(int fromIndex, int toIndex)** |
| | Returns a view of the portion of this list between the specified fromIndex, inclusive, and toIndex, exclusive. |
| **void** | **addFirst(E e)** |
| | Inserts the specified element at the beginning of this list. |
| **void** | **addLast(E e)** |
| | Appends the specified element to the end of this list. |
| **E** | **removeFirst()** |
| | Removes and returns the first element from this list. |
| **boolean** | **removeFirstOccurrence(Object o)** |
| | Removes the first occurrence of the specified element in this list (when traversing the list from head to tail). |
| **E** | **removeLast()** |
| | Removes and returns the last element from this list. |
| **boolean** | **removeLastOccurrence(Object o)** |
| | Removes the last occurrence of the specified element in this list (when traversing the list from head to tail). |
| **void** | **removeRange(int fromIndex, int toIndex)** |
| | Removes from this list all of the elements whose index is between fromIndex, inclusive, and toIndex, exclusive. |

## Possible practical exercises from the optional part

1. Determine what is the maximum depth of the bracket pairs (Task 1).
2. Find how many different pairs of parentheses there are (Task 1).
3. Find the minimum element of each subsequence bounded by the sliding window (Task 2).

   ...

## Possible theoretical issues

1. What is the computational complexity of the various operations implemented in the lab?
2. What are the advantages and disadvantages of the array-based and linked list-based stack implementation?
3. What are the advantages and disadvantages of the array-based and linked list-based queue implementation?
4. Be able to explain the various nuances involved in the programming code.

   ...