# Web Application 2

## Mode: Easy

### Exercise 1: Transpose

#### Arguments: Lists, Loops, Maps, Strings

Given an input text output it transposed.

Roughly explained, the transpose of a matrix:

ABC

DEF

is given by:

AD

BE

CF

Rows become columns and columns become rows.

If the input has rows of different lengths, this is to be solved as follows:

* Pad to the left with spaces.
* Don't pad to the right.

Therefore, transposing this matrix:

AB

DEF

results in:

AD

BE

F

In general, all characters from the input should also be present in the transposed output. That means that if a column in the input text contains only spaces on its bottom-most row(s), the corresponding output row should contain the spaces in its right-most column(s).

Implement the *Transpose* Kotlin object.

**object** Transpose {

**fun** **transpose**(input: List<String>): List<String> {

// **TODO:** implement me

}

}

### Exercise 2: Yacht

#### Arguments: Enums, Functional Programming, Lists, Pattern Matching

In the Yacht game, five dice are rolled and the result can be entered in any of twelve categories. The score of a throw of the dice depends on category chosen.

Scores in Yacht:

| **Category** | **Score** | **Description** | **Example** |
| --- | --- | --- | --- |
| Ones | 1 × number of ones | Any combination | 1 1 1 4 5 scores 3 |
| Twos | 2 × number of twos | Any combination | 2 2 3 4 5 scores 4 |
| Threes | 3 × number of threes | Any combination | 3 3 3 3 3 scores 15 |
| Fours | 4 × number of fours | Any combination | 1 2 3 3 5 scores 0 |
| Fives | 5 × number of fives | Any combination | 5 1 5 2 5 scores 15 |
| Sixes | 6 × number of sixes | Any combination | 2 3 4 5 6 scores 6 |
| Full House | Total of the dice | Three of one number and two of another | 3 3 3 5 5 scores 19 |
| Four of a Kind | Total of the four dice | At least four dice showing the same face | 4 4 4 4 6 scores 16 |
| Little Straight | 30 points | 1-2-3-4-5 | 1 2 3 4 5 scores 30 |
| Big Straight | 30 points | 2-3-4-5-6 | 2 3 4 5 6 scores 30 |
| Choice | Sum of the dice | Any combination | 2 3 3 4 6 scores 18 |
| Yacht | 50 points | All five dice showing the same face | 4 4 4 4 4 scores 50 |

If the dice do not satisfy the requirements of a category, the score is zero. If, for example, Four Of A Kind is entered in the Yacht category, zero points are scored. A Yacht scores zero if entered in the Full House category.

Given a list of values for five dice and a category, your solution should return the score of the dice for that category. If the dice do not satisfy the requirements of the category your solution should return 0. You can assume that the value of each will be between one and six inclusively. You should not assume that the dice are ordered.

Implement the *Yacht* Kotlin object.

**object** Yacht {

**fun** **solve**(category: YachtCategory,d1: Int, d2: Int, d3: Int, d4: Int, d5: Int): Int {

// **TODO:** implement me

}

**enum** **class** **YachtCategory** {

YACHT,

ONES,

TWOS,

THREES,

FOURS,

FIVES,

SIXES,

FULL\_HOUSE,

FOUR\_OF\_A\_KIND,

LITTLE\_STRAIGHT,

BIG\_STRAIGHT,

CHOICE

}

}

## Mode: Medium

### Exercise 3: Luhn

#### **Arguments**: Algorithms, Booleans, Loops, Strings, Type Conversion

Given a number determine whether or not it is valid per the Luhn formula.

The Luhn algorithm is a simple checksum formula used to validate a variety of identification numbers, such as credit card numbers and Canadian Social Insurance Numbers.

The task is to check if a given string is valid.

Strings of length 1 or less are not valid. Spaces are allowed in the input, but they should be stripped before checking. All other non-digit characters are disallowed.

**Example**

4539 1488 0343 6467

The first step of the Luhn algorithm is to double every second digit, starting from the right. We will be doubling.

4\_3\_ 1\_8\_ 0\_4\_ 6\_6\_

If doubling the number results in a number greater than 9 then subtract 9 from the product. The results of our doubling:

8569 2478 0383 3437

Then sum all of the digits:

8+5+6+9+2+4+7+8+0+3+8+3+3+4+3+7 = 80

If the sum is evenly divisible by 10, then the number is valid. This number is valid!

Implement the *Luhn* Kotlin object.

**object** Luhn {

**fun** **isValid**(candidate: String): Boolean {

// **TODO:** implement me

}

}

### Exercise 4: Kindergarten Garden

#### **Arguments**: Arrays, Enumerations, Lists, Logic, Loops, Pattern Recognitions, Strings

Given a diagram, determine which plants each child in the kindergarten class is responsible for.

The kindergarten class is learning about growing plants. The teacher thought it would be a good idea to give them actual seeds, plant them in actual dirt, and grow actual plants.

They've chosen to grow *grass*, *clover*, *radishes*, and *violets*.

To this end, the children have put little cups along the window sills, and planted one type of plant in each cup, choosing randomly from the available types of seeds.

[window][window][window]

........................ # each dot represents a cup

........................

There are 12 children in the class:

* Alice, Bob, Charlie, David
* Eve, Fred, Ginny, Harriet
* Ileana, Joseph, Kincaid, and Larry

Each child gets 4 cups, two on each row. Their teacher assigns cups to the children alphabetically by their names.

The following diagram represents Alice's plants:

[window][window][window]

VR......................

RG......................

In the first row, nearest the windows, she has a *violet* and a *radish*. In the second row she has a *radish* and some *grass*.

Your program will be given the plants from left-to-right starting with the row nearest the windows. From this, it should be able to determine which plants belong to each student.

Implement the *KindergartenGarden* class. The *diagram* should be in the following format: *"****VR\nRG****"*

**class** **KindergartenGarden**(**private** **val** diagram: String) {

**fun** **getPlantsOfStudent**(student: String): List<String> {

// **TODO:** implement me

}

}

### Exercise 5: Saddle Points

#### **Arguments**: Arrays, Conditionals, Exception Handling, Integers, Loops

Detect saddle points in a matrix.

So say you have a matrix like so:

1 2 3

|---------

1 | 9 8 7

2 | 5 3 2 <--- saddle point at column 1, row 2, with value 5

3 | 6 6 7

It has a saddle point at column 1, row 2.

It's called a "saddle point" because it is greater than or equal to every element in its row and less than or equal to every element in its column.

A matrix may have zero or more saddle points.

Your code should be able to provide the (possibly empty) list of all the saddle points for any given matrix.

The matrix can have a different number of rows and columns (Non square). *saddlePoints* should contains the set of saddle points, if exists, of a given matrix.

**data** **class** **MatrixCoordinate**(**val** row: Int, **val** col: Int)

**data** **class** **Matrix** (**val** matrix: List<List<Int>>) {

**val** saddlePoints: Set<MatrixCoordinate>

// **TODO:** implement me

}

### Exercise 6: Linked List

#### **Arguments**: Algorithms, Generics, Lists

Implement a doubly linked list.

Like an array, a linked list is a simple linear data structure. Several common data types can be implemented using linked lists, like queues, stacks, and associative arrays.

A linked list is a collection of data elements called nodes. In a singly linked list each node holds a value and a link to the next node. In a doubly linked list each node also holds a link to the previous node.

You will write an implementation of a doubly linked list. Implement a Node to hold a value and pointers to the next and previous nodes. Then implement a List which holds references to the first and last node and offers an array-like interface for adding and removing items:

* push (insert value at back);
* pop (remove value at back);
* shift (remove value at front).
* unshift (insert value at front);

**class** **Deque**<T> {

**fun** **push**(value: T) {

//TODO Implement me

}

**fun** **pop**(): T? {

//TODO Implement me

}

**fun** **unshift**(value: T) {

//TODO Implement me

}

**fun** **shift**(): T? {

//TODO Implement me

}

}

## Mode: Hard

### Exercise 7: List Ops

#### **Arguments**: Filtering, Functional Programming, Generics, Lists, Loops

Implement basic list operations. In functional languages list operations like *length*, *map*, and *reduce* are very common. Implement a series of basic list operations, without using existing functions.

The general operations you will implement include:

* append (given two lists, add all items in the second list to the end of the first list);
* concatenate (given a series of lists, combine all items in all lists into one flattened list);
* filter (given a predicate and a list, return the list of all items for which predicate(item) is True);
* length (given a list, return the total number of items within it);
* map (given a function and a list, return the list of the results of applying function(item) on all items);
* foldl (given a function, a list, and initial accumulator, fold (reduce) each item into the accumulator from the left using function(accumulator, item));
* foldr (given a function, a list, and an initial accumulator, fold (reduce) each item into the accumulator from the right using function(item, accumulator));
* reverse (given a list, return a list with all the original items, but in reversed order);

**val** List<Any>.customSize //TODO Implement me

**fun** <T> List<T>.**customAppend**(list: List<T>): List<T> {

//TODO Implement me

}

**fun** List<Any>.**customConcat**(): List<Any> {

//TODO Implement me

}

**fun** <T> List<T>.**customFilter**(predicate: (T) -> Boolean): List<T> {

//TODO Implement me

}

**fun** <T, U> List<T>.**customMap**(transform: (T) -> U): List<U> {

//TODO Implement me

}

**fun** <T, U> List<T>.**customFoldLeft**(initial: U, f: (U, T) -> U): U {

//TODO Implement me

}

**fun** <T, U> List<T>.**customFoldRight**(initial: U, f: (T, U) -> U): U {

//TODO Implement me

}

**fun** <T> List<T>.**customReverse**(): List<T> {//TODO Implement me}

### Exercise 8: Diffie Hellman

#### **Arguments**: Cryptography, Randomness, Algorithms, Integers

Alice and Bob use Diffie-Hellman key exchange to share secrets. They start with prime numbers, pick private keys, generate and share public keys, and then generate a shared secret key.

* Alice picks a private key, **a**, greater than 1 and less than *p*. Bob does the same to pick a private key **b**, supplying prime numbers **p** and **g**
* Alice calculates a public key **A**. Using the same *p* and *g*, Bob similarly calculates a public key **B** from his private key *b*.

A = g\*\*a mod p

* Alice and Bob exchange public keys. Alice calculates secret key **s**.

s = B\*\*a mod p

* Bob calculates

s = A\*\*b mod p

The calculations produce the same result!

Implement the *DiffieHellman* Kotlin object.

**object** DiffieHellman {

**fun** **privateKey**(prime: BigInteger): BigInteger {

//TODO Implement me

}

**fun** **publicKey**(p: BigInteger, g: BigInteger, privKey: BigInteger): BigInteger {

//TODO Implement me

}

**fun** **secret**(prime: BigInteger, publicKey: BigInteger, privateKey: BigInteger): BigInteger {

//TODO Implement me

}

}