Question 1: Complexity

Let N be a natural number. The following program takes as input a file `f` containing natural numbers below N (i.e., integers k satisfying $0 \le k < N$) and counts how many numbers are read before encountering a duplicate.

```
int longest_collision_free_prefix(File f):
    xs = new linked list
    i = 0
    repeat:
        read a number k from f
        if xs contains k:
            return i
        add k to xs
        i = i + 1
```

Assume that reading a number from the file never fails and is a constant time operation.

A. What is the worst-case asymptotic complexity of this program in N?
   Assume that reading a number from the file takes constant time.

B. What is the answer if we use a self-balancing binary search tree instead of a linked list?

Answer using O-notation. Your answer should be best-possible (sharp) and as simple as possible. Justify that the complexity of the program is of the stated order of growth.

(replace by answer)

Question 2: Sorting

Part A

In this exercise, you will define a *bottom-up* version of the merge sort algorithm for sorting lists. Instead of dividing a list in two and recursing until we reach a singleton (or empty) list, we start from the bottom and turn every element in the list into a singleton list. For example, the following list:

```
[2, 4, 3, 1, 5, 8, 9, 1]
```

is turned into a list of singletons as follows:

```
[[2], [4], [3], [1], [5], [8], [9], [1]]
```

In this list, the (initially) singleton lists are then pairwise merged into larger *sorted* lists in a number of iterations until there is only one list left. So, an iteration takes the first and second list and merges them, then it continues with the third and forth lists, and so on. (You may assume that the length of the input list is a power of two.) For instance, after one iteration the above list looks as follows:

```
[[2, 4], [1, 3], [5, 8], [1, 9]]
```

A list can be transformed into a list of singleton lists as follows in Java:

```java
List<List<Integer>> singularize(List<Integer> xs) {
    List<List<Integer>> xss = new LinkedList<>();
    for (Integer x : xs)
        xss.add(Collections.singletonList(x));
    return xss;
}
```

The implementation creates a singleton list for every element in `xs` and adds it to the result list of lists `xss`.

The next function merges two *already sorted* lists into one larger sorted list: (see next page)

```
List<Integer> merge(List<Integer> xs, List<Integer> ys) {
    List<Integer> res = new LinkedList<>();
    Iterator<Integer> ix = xs.iterator(), iy = ys.iterator();
    Integer x = next(ix), y = next(iy);
    while (x != null && y != null) {
        if (x <= y) { res.add(x); x = next(ix); }
        else        { res.add(y); y = next(iy); }
    }
    while (x != null) { res.add(x); x = next(ix); }
    while (y != null) { res.add(y); y = next(iy); }
    return res;
}

Integer next(Iterator<Integer> i) {
    return i.hasNext() ? i.next() : null;
}
```

Your task is to use the `merge` and `singularize` methods to implement a *bottom-up* version of merge sort. (You do not have to use Java, you can write in pseudocode.)

You **should not use indexed access** on lists (e.g., **not** `xs[k]`). You can iterate over lists as shown above or use lists as queues (e.g., `removeFirst()` and `addLast()`).

(replace by answer)

Part B

Consider the following list of numbers:

[1, 6, 4, 7, 9, 3, 5, 2]

Show how this list is sorted using bottom-up mergesort. Write down how the list is transformed after each iteration in the algorithm. In our example above we showed only one iteration, you need to show all iterations until there is only one (sorted) sublist left.

(replace by answer)

Question 3: Relaxed AVL trees

Suppose we use a binary search tree where the nodes are annotated with their *height*. In an ordinary AVL tree, the difference in height between the left and right child of each node is at most one. It is possible to relax this constraint and allow for a larger difference in height. This will result in slightly less balanced trees, but we don't need to rebalance that often. The tree rotations used to rebalance are the same as for an ordinary AVL tree.

In this question, we are going to allow for a **height difference of two** instead of one.

Consider the following sequence of integers:

8, 1, 7, 2, 3, 5

Starting with an empty tree, insert the numbers in the given order (using the natural ordering of integers). After each insertion, write down the resulting tree with each node annotated with its height. *Remember that we allow a height difference of two!*

(replace by answer)

Question 4: Priority queues

This question is about heaps, specifically **binary min-heaps** represented using arrays.

Part A

The following array represents a heap. However, some elements of the array have been *hidden* from you, by writing a "?" instead of the array element:

3	7	?	?	13	9	8	15	14	?

Replace the three "?"s with integers of your choice so that the resulting array is a valid heap. The array must also have **no duplicate elements**.

Write down the resulting heap, and explain briefly why the values that you chose work.

(replace by answer)

Part B

Pick an integer *x*, and add it to the heap from Part A, using the heap insertion algorithm. However, after the insertion, *x* must end up at **index 1** of the array (`arr[1]` in Java). Assume that array indices start from 0.

Also, just as in Part A, the array must have **no duplicate elements** after the insertion.

Write down:

- the integer *x* that you chose,
- the final heap, written as an array,
- a brief explanation of **why** *x* ended up at that position (one sentence is enough).

(replace by answer)

Question 5: Hash tables

Part A: Separate chaining

For Part A, use the following pairs of keys and hash values:

G:12, E:9, A:8, F:11, B:5, D:14, C:5

Assumptions:

1. We use separate chaining with linked lists.
2. The size of the hashtable is m = 5 and we use modular compression.

We start with an empty hash table and insert the given keys **in the order given above** (i.e. G:12 first and C:5 last). For each table below that **cannot** be constructed in this way, explain briefly why it is impossible.

Hash table 1	
Index	Key(s)
0	B C
1	F
2	G
3	A
4	E D

Hash table 2	
Index	Key(s)
0	B C
1	F
2	G
3	A
4	E D

Hash table 3	
Index	Key(s)
0	C B
1	F
2	G
3	A
4	D E

Hash table 4	
Index	Key(s)
0	C B
1	F
2	G
3	A
4	D E

(replace by answer)

Part B: Open addressing

Use the following pairs of keys and hash values:

B:10, C:10, A:11, F:11, D:12, G:13, E:14

Assumptions:

1. We use open addressing with linear probing to resolve collisions (with probing constant 1, i.e. h’ = h + 1).
2. The size of the hashtable is m = 8 and we use modular compression.

We start with an empty hash table and insert the given keys **in an unspecified order**.

For each table below that **cannot** be constructed with the assumptions above, explain briefly why it is impossible.

For each table below that **can** be constructed, give an insertion order that will yield the table.

Hash table 1	
Index	Key
0	G
1	
2	B
3	C
4	F
5	D
6	A
7	E

Hash table 2	
Index	Key
0	E
1	
2	C
3	A
4	F
5	B
6	D
7	G

Hash table 3	
Index	Key
0	G
1	
2	C
3	B
4	D
5	E
6	A
7	F

Hash table 4	
Index	Key
0	D
1	
2	B
3	C
4	G
5	A
6	E
7	F

(replace by answer)

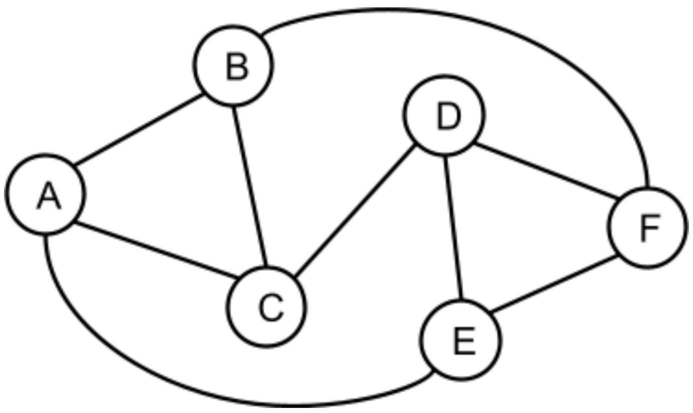
Question 6: Graphs

Part A

Some time ago you ordered an ***undirected*** and ***weighted*** graph from the company https://graphazon.com, and you had some specific requirements that the graph should satisfy:

- There must be exactly 6 nodes, called A, B, C, D, E and F.
- There must be exactly 9 edges, and all of them must have distinct edge costs.
- The cheapest path between A and F must cost exactly 23.
- The cost of the minimum spanning tree (MST) must be exactly 39.

Now finally, after several weeks of eager waiting, the graph has arrived:



Unfortunately, all the weights got lost during transport! After some searching you finally found them hiding at the bottom of the box:

3, 7, 8, 9, 12, 23, 26, 28, 29

Now you have to match each edge with a weight so that the final graph satisfies the constraints above. If you succeed with that, the Graphazon company has hinted that they might be interested in hiring you as a graph building consultant.

List here each edge with its weight you assigned (you can e.g. write an edge as DF or D–F).

(replace by answer)

List the edges (with weights) that form the cheapest path from A to F. Explain why there is no cheaper path and why its cost is 23.

(replace by answer)

Part B

Run ***Prim's algorithm*** on your final graph, hopefully resulting in an MST with cost 39.

Answer by listing each edge in the order it is added to the MST by Prim's algorithm. Also, explain why the MST has cost 39.

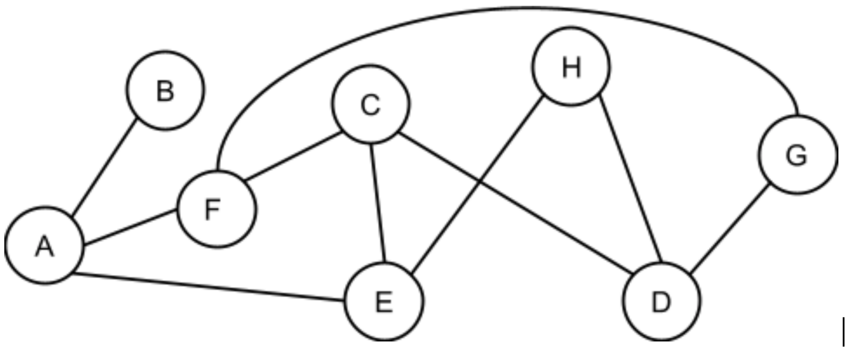
(replace by answer)

Question 7 (advanced): The two-colouring problem

Let's call an undirected, connected graph "two-colourable", if it is possible to partition its nodes into two disjoint subsets so that there are no edges that go between nodes in the same subset. Another way of viewing this is that you can colour every node either ***Indigo*** or ***Olive*** so that no edge goes between nodes of the same colour.

Warming up

Here is an example of a two-colourable graph:



Partition the nodes into the sets ***Indigo*** and ***Olive***, to show that the graph indeed is two-colourable:

Indigo = …

Olive = …

(Note that you will not get any points if you only answer this warming-up question)

Part A

Design an efficient algorithm that takes a graph as input and returns the set of ***Indigo*** nodes if and only if the graph is two-colourable. If it's not, it should return **null**.

Don't forget to describe which data structures you use to implement (1) the graph, (2) the ***Indigo*** set, and (3) possibly other intermediate variables.

(replace by answer)

Part B

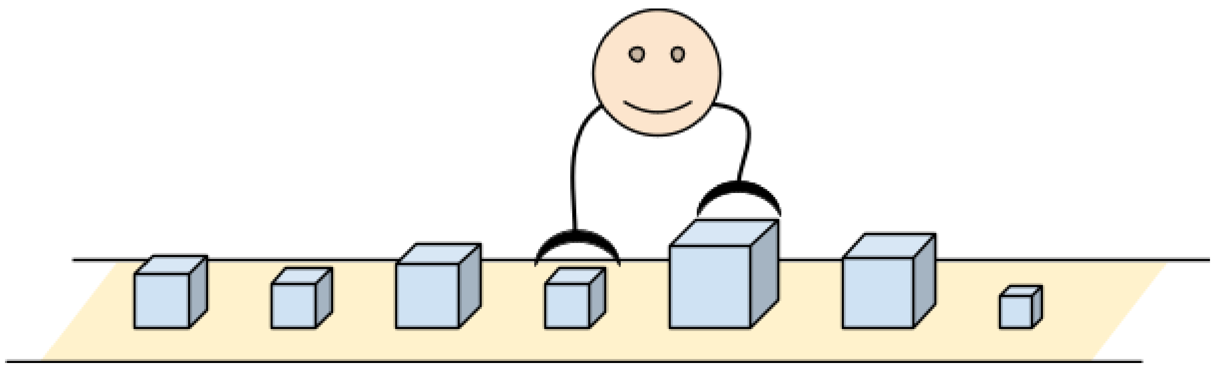
Which is the worst-case time complexity of your algorithm, in terms of the number of nodes ***V*** and the number of edges ***E***?

Answer using O-notation. Your answer should be best-possible (sharp) and as simple as possible. Briefly justify that the complexity of your algorithm is of the stated order of growth.

(replace by answer)

Question 8 (advanced): Sorting robot

Assume you have a long line of *N* objects of different sizes, and a robot:



You want to instruct the robot to sort the long line (with the smaller objects to the left), but it has just a limited instruction set. The robot is always in front of two adjacent objects (as in the picture). The number of objects is known (called *N*) and the robot always starts in the leftmost position. The robot understands the following instructions:

- LEFT: move one step left (if possible),
- RIGHT: move one step right (if possible),
- COMPARE: compare the two objects in front of it, returning true if the left object is smaller than the right object,
- SWAP: swap places of the two objects in front of it.

Part A

Implement an algorithm that makes the robot sort the objects.

(replace by answer)

Part B

What is the asymptotic complexity of the algorithm in *N*? Briefly justify.

(replace by answer)

Question 9 (advanced): A most peculiar function

We have a peculiar function, which in pseudo-code looks like this:

```
// Assume n >= 0
int peculiar(int n):
    acc = 0
    if n == 0:
        acc = 1
    else
        for i = 1 to n:
            n1 = floor(i / 3)
            n2 = i - 2
            acc = acc + peculiar(n1) + peculiar(n2)
    return acc
```

The floor-function rounds down, e.g. floor(1) = 1, floor(1.9) = 1, floor(-0.5) = -1.

One problem with the function is that it is very slow to compute; it takes exponential time in *n*. (The value of the function itself grows quite rapidly, starting at *n* = 0, {1, 2, 4, 8, 14, 24, 42, 70, …}.)

Assume that all arithmetic operations and floor(x) take constant time.

Part A

Rewrite the function so that its time complexity is quadratic, O(n^2), and space complexity is linear, O(n), in the input argument *n*. It should of course still give the same result as the original function!

(replace by answer)

Part B

Briefly justify why the time complexity is O(n^2) and space complexity is O(n).

(replace by answer)