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?

B. What is the answer if we replace the linked list by a self-balancing binary search tree?

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.

The program consists of a single loop, so our first job is to find out how many iterations it can have before stopping because the return statement is executed. Since there are exactly N natural numbers below N, we are guaranteed to find a duplicate after reading at most N + 1 numbers. So the loop executes at most N + 1 times, which is O(N) (this also describes the size of `xs`). So the total asymptotic complexity is N times that of the loop body.

For Part A, the loop body is O(N) because `xs contains k` is linear in the size of `xs` (the rest of the loop is constant time). So the total complexity is $O(N^2)$.

For Part B, the loop body is O(log(N)) because `xs contains k` and `add k to xs` are logarithmic in the size of `xs`. So the total complexity is O(N log(N)).

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:

```
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()`).

A solution using iterators:

```
List<Integer> bottom_up_merge_sort(List<Integer> xs):
    xss = singularize(xs)
    while xss.size() > 1:
        Iterator<LinkedList<Integer>> it = xss.iterator()
        xss = new LinkedList<List<Integer>>()
        while it.hasNext():
            xss.add(merge(it.next(), it.next()))
    return xss.getFirst()
```

A solution using queue methods:

```
List<Integer> bottom_up_merge_sort(List<Integer> xs):
    xss = singularize(xs)
    while xss.size() > 1:
        xss_new = LinkedList<List<Integer>>()
        while not xss.isEmpty():
            xss_new.add(merge(xss.removeFirst(), xss.removeFirst()))
        xss = xss_new
    return xss.getFirst()
```

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 of lists 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) list left.

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

The final sorted list is [1, 2, 3, 4, 5, 6, 7, 9].

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. Use the natural ordering of integers in the binary search tree. After each insertion, write down the resulting tree with each node annotated with its height. *Remember that we allow a height difference of two!*

We follow the insertion procedure for AVL trees: we first do BST insertion (using the natural ordering of the integers), then go upwards from the new leaf node to the root, examine the height invariant for each node encountered, and use rotations to rebalance if necessary. The only difference is that rebalancing happens if the height difference between the left and right child of the current node is more than **two**.

After inserting 8:

```
8 [height 0]
```

After inserting 1:

```
┌── 1 [height 0]
8 [height 1]
```

After inserting 7:

```
┌── 1 [height 1]
│   └── 7 [height 0]
8 [height 2]
```

After inserting 2, we have to rebalance at node 8 (left-right case):

```
┌── 1 [height 1]
│   └── 2 [height 0]
7 [height 2]
└── 8 [height 0]
```

After inserting 3:

```
┌── 1 [height 2]
|    └── 2 [height 1]
|         └── 3 [height 0]
7 [height 3]
└── 8 [height 0]
```

After inserting 2, we have to rebalance at node 1 (right-right case):

```
     ┌── 1 [height 0]
┌── 2 [height 2]
|    └── 3 [height 1]
|         └── 5 [height 0]
7 [height 3]
└── 8 [height 0]
```

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.

Here is one possible solution:

3	7	6	11	13	9	8	15	14	16

The value for each "?"s needs to be chosen so that it is greater than its parent and less than its two children (and so that it is unique). To determine the parents and children, one can draw the heap as a tree or use the formulas for computing (e.g. using 0-based indexing) from index i the index (i-1) / 2 of the parent and the indices 2i + 1 and 2i + 2 of the children.

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).

To add an element to a heap, we add it to the end of the array and then move ("swim") the element up as long as it is smaller than its parent. So to end up at index 1, it must be smaller than the current element 7 at that index, but larger than its parent 3. It must also not already occur in the table. For example, we can take x = 5.

3	5	6	11	7	9	8	15	14	16	13

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 hashing.

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

All elements are in the slot corresponding to the element's hash value % 5.

There are two slots with two keys each. If we use a linked list, elements are added at the front or at the end of the list **but not both**.

The intent was to have four distinct tables but in some cases there were only two.

The table with contents [[[B, C], [F], [G], [A], [E, D]]] is possible -- elements are consistently added first or last.

The table with contents [[[C, B], [F], [G], [A], [D, E]]] is possible -- elements are consistently added first or last.

Part B: Open addressing

Use the following pairs of keys and hash values:

B:10, C:10, F:11, A: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 hashing.

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

Table 1 can be generated with this insertion order: [B:10, C:10, F:11, D:12, A:11, E:14, G:13]

Table 2 can be generated with this insertion order: [C:10, A:11, F:11, B:10, D:12, G:13, E:14]

Table 3: E has hash value 14 (= 6 modulo 8), but is in slot 5. This is impossible because there is an empty slot, 1, in between that would have been probed during insertion.

Table 4: G has hash value 13 (= 5 modulo 8), but is in slot 4. This is impossible because there is an empty slot, 1, in between that would have been probed during insertion.

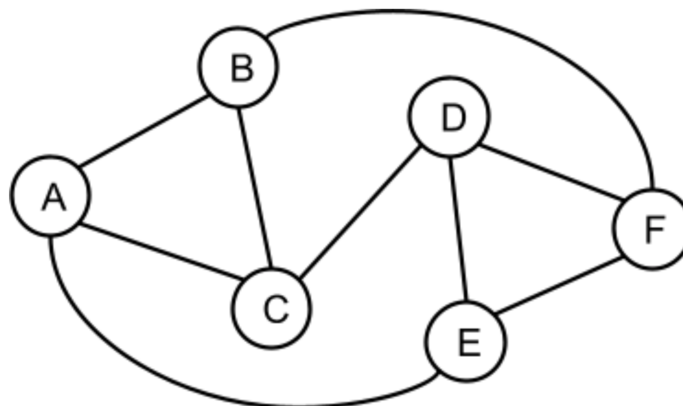
Question 6: Graphs

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).

One possible solution: AB:23, AC:3, AE:7, BC:26, BF:9, CD:8, DE:28, DF:12, EF:29.

Here is one strategy (out of many) to find such a solution. (This is not a required part of the answer and only provided for illumination.)

Every path from A to F has at least two edges. To get a cost of 23 for such a path, we cannot use the weights 23 and higher. That leaves the first five weights. By trying combinations, one sees that 3, 8, 12 sum to 23. So we use these weights for some path of length 3 from A to F (here, A-C-D-F).

Since there are 6 nodes, any spanning tree consists of 5 edges. The sum of the 5 cheapest weights is already 39, so they should form the MST. We already have the path A-C-D-F, so

we add 2 more edges (here, AE and BF) to get a spanning tree and assign the weights 7 and 9 to them.

The remaining 4 (expensive) weights we assign arbitrarily. Since there is a unique path between any two nodes in a tree, every path from A to F other than the one above needs to cross one of them, so it cannot cost less than 23.

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.

The cheapest path is A --[3]-- C --[8]-- D --[12]-- F. Its cost is the sum of weights, which is indeed 23.

Why is there no cheaper path from A to F? In any other way of going from A to F, we need to cross one of the expensive edges AB:23, BC:26, DE:28, EF:29. These alone make the path cost at least 23.

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.

We choose A as the starting node for Prim's algorithm (we could choose any other node as well; the computed MST will be the same).

The algorithm adds edges to the MST in the following order: AC:3, AE:7, CD:8, BF:9, DF:12.

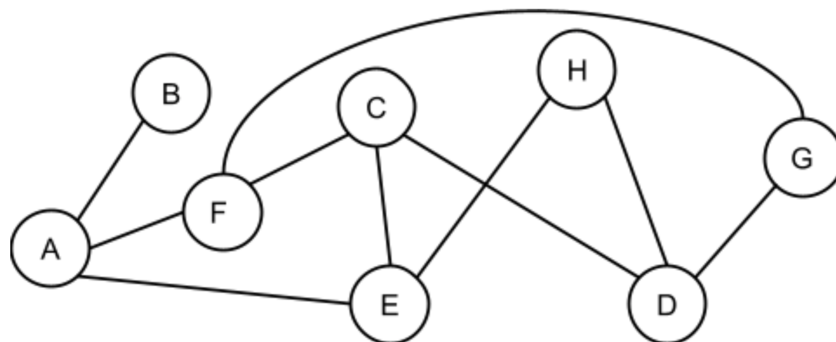
The sum of the weights of these edges is 39, which is thus the cost of the MST.

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 = {A, C, G, H}

Olive = {B, F, E, D}

(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.

There are many possible solutions. We use adjacency lists to implement the graph and (e.g.) an AVL tree to implement the sets of Indigo and Olive nodes.

We start with an arbitrary node and colour it Indigo. We then perform a depth-first search starting from this node. Whenever we discover a new node, we colour it with the opposite colour used for the node from which it is reached via an edge.

```
// Recursively colour the nodes,
// starting by putting node in colour_0.
```

```
def void two_colour(Graph graph, Node node,
                    Set<Node> colour_0, Set<Node> colour_1):
    // Test if node has been coloured with the wrong colour before.
    if colour_1 contains node:
        raise error "graph is not two-colourable"

    // Only proceed if node has not already been coloured.
    if colour_0 contains node:
        colour_0.add(node)
        for edge in graph.outgoingEdges(node):
            // Swap colours by exchanging colour_0 and colour_1.
            two_colour(graph, edge.target(), colour_1, colour_0)

def Set<Node> compute_indigo_nodes(Graph graph, Node starting_node):
    indigo = new set of nodes
    olive = new set of nodes
    two_colour(graph, starting_node, indigo, olive)
    return indigo
```

Part B

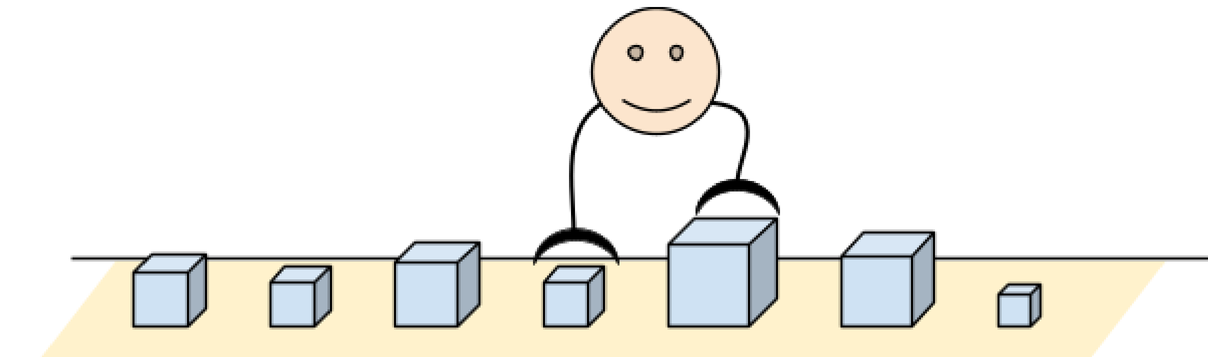
Which is the worst-case time complexity of your algorithm, in terms of the number of nodes ***V*** and the number of edges ***E***?

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.

There will be $2E+1 \in O(E)$ calls to `two_colour`: the initial call and two calls for every edge. In every call, we have some constant time statements, some checks if node is in `colour_0` or `colour_1` with complexity $O(\log(V))$, `colour_0.add(node)` with complexity $O(\log(V))$, and a for-loop, but the complexity of the for-loop is already accounted for since all it does is call two_colour again. In total, the complexity is $O(E \log(V))$.

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, returning true if the left object is smaller than the right object,
- SWAP: swap places of the two objects in front.

Part A

Implement an algorithm that makes the robot sort the objects.

There are many possible solutions. Here is one based on bubble sort:

```
repeat:
    // Move to the left end.
    loop N-1 times: LEFT

    // Go from left to right and swap whenever
    // adjacent elements are in the wrong order.
    in_order = True
    loop N-1 times:
        if not COMPARE:
            In_order = False
            SWAP
        RIGHT

    if in_order: break // break out of loop
```

After k iterations, the k largest elements will be in the correct place. So the outer loop iterates at most N times.

Part B

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

The outer loop has O(N) iterations. All inner loops have O(N) iterations. Everything else is constant time. So the total complexity is O(N^2).

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!

The key idea is to store the values computed by `peculiar` in a lookup table and access it before (re)computing a value. This technique is called memoization.

In this case, we can use a simple array as a lookup table. In general, one can use any data structure implementing a map from arguments to computed values.

```
int peculiar(int n):
    return peculiar_memoized(n, new Integer[n+1])
    // We assume the array is initialized with “null” values.

int peculiar_memoized(int n, Integer[] values):
    if n < 0: return 0 // should have been specified in question

    // First try to look the value up in the lookup table.
    // If we find a value, return it.
    if values[n] != null:
        return values[n]

    acc = 0
```

```
    if n == 0:
        acc = 1
    else:
        for i = 1 to n:
            n1 = floor(i / 3)
            n2 = i - 2
            acc = acc + peculiar_memoized(n1, values)
                      + peculiar_memoized(n2, values)

    // Store the computed value in the lookup table.
    values[n] = acc
    return acc
```

With more thought, one can optimize even more:

```
int peculiar_memoized(int n, Integer[] values):
    if n < 0: return 0 // should have been specified in question

    // First try to look the value up in the lookup table.
    // If we find a value, return it.
    if values[n] != null:
        return values[n]

    if n == 0:
        acc = 1
    else:
        acc = peculiar_memoized(n - 1, values)
            + peculiar_memoized(floor(n / 3), values)
            + peculiar_memoized(n - 2, values)

    // Store the computed value in the lookup table.
    values[n] = acc
    return acc
```

Part B

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

When we call `peculiar(n)`, every value for i from 0 to n is computed at most once (it is then stored in the lookup table and just taken from there every time it is needed afterwards). This computation is a loop with O(n) iterations and some constant time operations (assuming that the values for j less than i have already been computed), so is O(n). There are O(n) possibilities for i, so the total complexity is O(n^2).

In the second solution given, the loop in the computation of `peculiar_memoized(n)` has been eliminated. Disregarding the recursive calls, it is O(1). So the total complexity is O(n).

The lookup table grows to at most n+1 elements (and so does the amount of nested function calls), so the space complexity is O(n).

Very advanced question: find an implementation with time complexity O(n) and space complexity O(log(n)).